

使用Python和Selenium进行Web自动化测试的实战指南

Selenium自动化测试

——基于Python语言

Learning Selenium Testing
Tools with Python

[印度] 昂米沙·冈迪察 (Unmesh Gundecha) 著

金鑫 熊志男 译

testwo测试窝 审

目 录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[推荐序](#)

[译者序](#)

[业界评价](#)

[关于作者](#)

[作者语](#)

[关于审稿人](#)

[前言](#)

[第1章 基于Python的Selenium WebDriver入门](#)

[1.1 环境准备](#)

[1.1.1 安装Python](#)

[1.1.2 安装Selenium包](#)

[1.1.3 浏览Selenium WebDriver Python文档](#)

[1.1.4 选择一个IDE](#)

[1.1.5 PyCharm设置](#)

[1.2 第一个Selenium Python脚本](#)

[1.3 支持跨浏览器](#)

[1.3.1 设置IE浏览器](#)

[1.3.2 设置Google Chrome浏览器](#)

[1.4 章节回顾](#)

[第2章 使用unittest编写单元测试](#)

[2.1 unittest单元测试框架](#)

[2.1.1 TestCase类](#)

[2.1.2 类级别的setUp\(\)方法和tearDown\(\)方法](#)

[2.1.3 断言](#)

[2.1.4 测试套件](#)

[2.2 生成HTML格式的测试报告](#)

[2.3 章节回顾](#)

[第3章 元素定位](#)

[3.1 借助浏览器开发者模式定位](#)

[3.1.1 用火狐浏览器Firebug插件检查页面元素](#)

[3.1.2 用谷歌Chrome浏览器检查页面元素](#)

[3.1.3 用IE浏览器检查页面元素](#)

[3.2 元素定位](#)

[3.2.1 ID定位](#)

- [3.2.2 name定位](#)
- [3.2.3 class定位](#)
- [3.2.4 tag定位](#)
- [3.2.5 XPath定位](#)
- [3.2.6 CSS选择器定位](#)
- [3.2.7 Link定位](#)
- [3.2.8 Partial link定位](#)
- [3.3 方法实践](#)
- [3.4 章节回顾](#)

[第4章 Selenium Python API介绍](#)

- [4.1 HTML表单元素](#)
- [4.2 WebDriver原理](#)
 - [4.2.1 WebDriver功能](#)
 - [4.2.2 WebDriver方法](#)
- [4.3 WebElement接口](#)
 - [4.3.1 WebElement功能](#)
 - [4.3.2 WebElement方法](#)
- [4.4 操作表单、文本框、复选框、单选按钮](#)
 - [4.4.1 检查元素是否启用或显示](#)
 - [4.4.2 获取元素对应的值](#)
 - [4.4.3 is_selected\(\)方法](#)
 - [4.4.4 clear\(\)与send_keys\(\)方法](#)
- [4.5 操作下拉菜单](#)
 - [4.5.1 Select原理](#)
 - [4.5.2 Select功能](#)

[4.5.3 Select方法](#)

[4.6 操作警告和弹出框](#)

[4.6.1 Alert 原理](#)

[4.6.2 Alert功能](#)

[4.6.3 Alert方法](#)

[4.6.4 浏览器自动化处理](#)

[4.7 章节回顾](#)

[第5章 元素等待机制](#)

[5.1 隐式等待](#)

[5.2 显式等待](#)

[5.3 expected_conditions类](#)

[5.3.1 判断某个元素是否存在](#)

[5.3.2 判断是否存在Alerts](#)

[5.4 预期条件判断的实践](#)

[5.5 章节回顾](#)

[第6章 跨浏览器测试](#)

[6.1 Selenium Standalone Server](#)

[6.1.1 下载Selenium Standalone Server](#)

[6.1.2 启动Selenium Standalone Server](#)

[6.2 在Selenium Standalone Server上执行测试](#)

[6.2.1 配置IE支持](#)

[6.2.2 配置Chrome支持](#)

[6.3 Selenium Grid](#)

[6.3.1 启动hub](#)

[6.3.2 添加节点](#)

[6.4 Mac OS X的Safari节点](#)

[6.5 在Grid上执行测试](#)

[6.6 在云端执行测试](#)

[6.7 章节回顾](#)

[第7章 移动端测试](#)

[7.1 认识Appium](#)

[7.1.1 Appium支持的应用类型](#)

[7.1.2 Appium环境准备](#)

[7.2 安装Appium](#)

[7.3 iOS测试](#)

[7.4 Android测试](#)

[7.5 使用Sauce Labs](#)

[7.6 章节回顾](#)

[第8章 Page Object与数据驱动测试](#)

[8.1 数据驱动测试](#)

[8.2 使用ddt执行数据驱动测试](#)

[8.2.1 安装ddt](#)

[8.2.2 设计一个简单的数据驱动测试](#)

[8.3 使用外部数据的数据驱动测试](#)

[8.3.1 通过CSV获取数据](#)

[8.3.2 通过Excel获取数据](#)

[8.4 Page Object设计模式](#)

[8.4.1 测试准备](#)

[8.4.2 BasePage对象](#)
[8.4.3 实现Page Object](#)
[8.4.4 构建Page Object模式测试实例](#)
[8.5 章节回顾](#)

[第9章 Selenium WebDriver的高级特性](#)

[9.1 键盘与鼠标事件](#)
[9.1.1 键盘事件](#)
[9.1.2 鼠标事件](#)
[9.2 调用JavaScript](#)
[9.3 屏幕截图](#)
[9.4 屏幕录制](#)
[9.5 弹出窗的处理](#)
[9.6 操作cookies](#)
[9.7 章节回顾](#)

[第10章 第三方工具与框架集成](#)

[10.1 行为驱动开发（BDD）](#)
[10.1.1 Behave安装](#)
[10.1.2 第一个feature](#)
[10.2 持续集成Jenkins](#)
[10.2.1 Jenkins环境准备](#)
[10.2.2 搭建Jenkins](#)
[10.3 章节回顾](#)

[欢迎来到异步社区！](#)

版权信息

书名：Selenium自动化测试——基于 Python 语言

ISBN：978-7-115-46174-2

本书由人民邮电出版社发行数字版。版权所有，
侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，
未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们
共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包
括但不限于关闭该帐号等维权措施，并可能追究法

律责任。

• 著 Unmesh Gundecha

译 熊志男

责任编辑 张 涛

• 人民邮电出版社出版发行 北京市丰台区成
寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

• 读者服务热线: (010)81055410

反盗版热线: (010)81055315

版权声明

Copyright © Packt Publishing 2014. First published in the English language under the title“Learning Selenium Testing Tools with Python”, ISBN 978-1-78398-350-6.

All rights reserved.

本书中文简体字版由**Packt Publishing**公司授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

内容提要

Selenium是一个主要用于Web应用程序自动化测试的工具集合，在行业内已经得到广泛的应用。本书介绍了如何用Python语言调用Selenium WebDriver接口进行自动化测试。主要内容为：基于Python的Selenium WebDriver入门知识、第一个Selenium Python脚本、使用unittest编写单元测试、生成HTML格式的测试报告、元素定位、Selenium Python API介绍、元素等待机制、跨浏览器测试、移动端测试、编写一个iOS测试脚本、编写一个Android测试脚本、Page Object与数据驱动测试、Selenium WebDriver的高级特性、第三方工具与框架集成等核心技术。

本书适合任何软件测试人员阅读，也适合作为

大专院校师生的学习用书和培训学校的教材。

推荐序

认识熊志男是在中国质量大会BQConf的活动上，交谈间很快就被志男对于测试领域的见解和趋势展望所折服，而这不仅仅是因为他有丰富的测试经验，而且还因为他作为测试窝的联合创始人，对国内外测试行业有深入的了解密不可分。本书正是一个例证。

尽管国内测试行业宣扬自动化测试已经很多年了，但是我们很难在自动化测试领域招聘到经验丰富的工程师，这说明自动化测试并没有成为国内测试领域的主流。

是因为测试人员不够努力导致的吗？我并不这么认为。

很多公司对于测试的投入是希望知道产品有多少缺陷、能否按时上线，所以相应的测试人员的工作都聚焦于如何高效地编写和执行测试用例，而自动化测试并不是第一选择。因为在上线压力巨大的情况下，如果不能评估出自动化测试的投入产出比，很难让项目经理在自动化测试上进行投入。所以倘若有实践自动化测试的想法，往往需要测试人员付出自己的时间来熟悉框架、编写和维护自动化测试。而如果不是事先对测试框架有了比较深入的认识，依靠自身自发进行自动化测试，并不会带来效率的明显提升。这样不仅会让项目经理失去信心，恐怕测试人员自己也心存疑虑了。

如果单从技术上考虑，究竟是什么阻碍着测试人员广泛使用自动化测试呢？首先，如果没有一个通用的测试框架，那么每做一个项目，测试人员就得学习一套新的工具 / 框架，这样的学习成本太高了。在工期很紧的情况下更是如此。其次，自动化

测试的编写实际上是进行编码，如果使用Java和C#这些编程语言编写自动化测试，由于测试人员很难全面掌握这些语言的开发技巧，容易导致编写出的自动化测试代码比产品代码出现更多的缺陷。

本书直击这两方面，为测试人员解除了后顾之忧。

（1）Selenium WebDriver作为业界通用的测试框架，不仅是Web测试的标准，在移动测试领域也是底层的核心驱动框架。所以掌握了Selenium WebDriver，可以让我们在为Web产品和移动产品编写自动化测试时游刃有余。

（2）Python作为动态语言，简化了严格的编程语法，使测试人员更容易掌握。同时Python也提供了丰富的API和扩展，测试人员可以很便利地调用或者集成其他语言编写的程序和类库，提高编写自动化测试的效率。

本书在讲述自动化测试编写的同时，结合业界主流的自动化测试开发模式，向读者介绍了多种测试相关知识（如BDD和持续集成）。非常推荐对测试有激情，希望快速提升自动化测试能力的朋友阅读本书。

黄 勇

现任ThoughtWorks中国区QA Lead

译者序

起初接到本书的翻译邀约时，内心还是有一些困惑的。针对软件测试行业，特别是基于Web自动化测试领域，Selenium已经是广泛使用的工具之一了，而且已被诸多测试同行认可并使用。为此，我们查阅了国内大量相关书籍或文章，发现当前Selenium的初学门槛其实并不高，测试工程师具备有功能测试经验，加之以对Web前端技术的一定程度的理解，外加较熟练地掌握一门脚本语言，经过一段时间的项目锤炼，都能基本应对日常自动化测试任务。不过与此同时，我们也发现很多初学者遇到的诸多困惑，又或者在深入学习过程中难以克服的瓶颈。

行业内，能系统性介绍WebDriver原理、多类

型Server运行方式、单元测试以及如何使用Python调用Selenium WebDriver接口的具体实例的材料相对零散。直到《*Learning Selenium Testing Tools with Python*》中文版的出版，使得我们有机会较为全面，并且系统性地学习用单一脚本语言开发Web自动化测试的具体实践，作者独特的创作逻辑，使得本书前后实例相互对照，并且首尾呼应。既诠释原理，又能使读者进入实战，还有“干货”满满的“提醒与备注”，是一本不可多得的自动化测试指导书。这也是我们译著这本书最重要的原因了。

本书的作者Unmesh Gundecha有着极为丰富构建自动化测试解决方案的经验。主导开发过大量商业或开源的自动化测试工具。曾供职微软。在2012年编著过《*Selenium Testing Tools Cookbook*》一书，颇为畅销；在2015年下半年又更新发布了第二版。作者文笔犀利，逻辑之间环环相扣，语言诙谐，妙笔生花。

多年的技术文章翻译经验，使我们清晰地认识到，倘若停留在专业翻译层面，想必本书的可读性，以及作者的诸多表述，都难以顺利地传递给中文读者。所以我与熊志男（本书合译者），多次调整翻译策略。由传统的分章节翻译，到“按作者编著脉络”组织分工。由专业词汇翻译，到统一关键词口径，甚至到整句、整段打乱重组。诸如此类的一些做法，都是为了保证我们的译著质量更加符合测试同行的阅读习惯，便于学习与加深印象。

好在翻译过程中，与熊志男相互鼓励，包括审核团队不厌其烦的讨论、PK、一起揣摩作者意图。大家的专业、包容、豁达、自信，让这本书的中文译著得以完成。特别感谢参与翻译工作的张欣欣、谢满彬、谢柳娜。感谢测试窝网译文团队的多次审校。

翻译别人的图书，好似在反刍，再精彩也是在

讲别人的故事。期待有一天，有机会能够讲讲我们自己的故事给广大同行。由于译者的水平有限，难免会有偏差疏漏。若有欠妥之处，欢迎指正，编辑联系邮箱zhangtao@ptpress.com.cn。

金 鑫

业界评价

在互联网行业迅速发展的今天，编写自动化脚本的技能，已经逐渐成为 Web 测试人员的标配。

Python 作为备受测试人员青睐的语言之一，非常适合处理日常工作中的数据和文本问题。

Selenium 更是 UI 自动化测试的利器，但要迅速掌握并熟练运用到项目中，绝非易事。

本书围绕 Selenium 的使用展开，编排有序，通俗易懂，对于没有 UI 自动化测试经验的读者，将起到事半功倍的效果。

——Ping++ 质量负责人 吴子腾

Unmesh Gundecha编著的《*Selenium Testing*

Tools Cookbook》，俗称“Selenium菜谱”，是我一直推荐给身边WebDriver初学者的书籍，只是很遗憾一致未被翻译成中文版本出版。

该书作为“菜谱”的Python姊妹篇，秉承了“菜谱”的内容详实，案例丰富，行文流畅等特点，是一本WebDriver入门的绝佳教材。

——陈冬严，浙江大学硕士，具有10年软件测试和团队管理的工作经验，先后服务于领先的ITSM、PLM软件研发企业，现于某金融行业核心机构IT规划部门担任项目管理工作。业余时间喜欢园艺。《精通自动化测试框架设计》一书的作者。

关于作者

Unmesh Gundecha拥有计算机软件硕士学位，在软件开发与测试领域有着12年的工作经验。无论是在应对业界标准，还是定制需求下，他都有着丰富的构建自动化测试解决方案的经验。与此同时，他还主导开发了大量商业或开源的自动化测试工具。

他曾供职于微软公司，从事开发有关的工作。目前在印度的一家跨国企业从事测试架构师工作，在Ruby、Java、iOS、Android和PHP的项目中有着极丰富的开发与测试经验。

作者语

另外，本书能顺利编写完成，离不开很多技术同行的帮助与审阅，感谢他们花费了大量的时间为本书提供了非常有价值的反馈。

感谢各位专家、同事和朋友，特别是Yuri Weinstein给予我很多帮助与鼓励。

关于审稿人

Adil Imroz是一位Python的狂热爱好者，长期专注在测试开发与移动端自动化领域。崇尚开源与敏捷模式。闲暇时，爱好单车、读书、睡觉。他觉得这些都可以为他开拓眼界。

Dr. Philip Polstra（熟悉他的人都称呼他**Dr. Phil**），国际知名黑客。他的作品曾在许多国际的专题会议（包括DEFCON、Black Hat、44CON、Maker Faire等）上提及，发表过大量的论文，是这一领域公认的专家。

Dr. Polstra作为布鲁斯伯格大学的副教授，除了日常教学，还对外提供一些关于渗透测试的培训、咨询工作。

Walt Stoneburner，软件架构师，在商业应用开发与咨询领域有着25年以上的经验，另外在软件质量保证、配置管理与安全领域也有着长期的研究。

无论是在程序设计、协作应用、大数据、知识管理、数据可视化，还是在ASCII方面，他都有着很深的造诣。甚至在软件评测、消费电子产品测评、绘画、经营摄影工作室、创作幽默剧、游戏开发、无线电等领域都能找到他的身影，他还自称“极客”。

Yuri Weinstein，生活在旧金山，有超过20年的时间就职于硅谷顶尖的技术公司，专注测试领域，尤其是在自动化测试方向。目前在红帽公司负责Ceph开源存储项目的产品质量。

前言

Selenium是一个主要用于Web应用程序自动化测试的工具集合，在行业内已经得到广泛的应用。然而其作用不局限于测试领域，还可以用于屏幕抓取与浏览器行为模拟等操作。它支持主流的浏览器，包括Firefox、IE、Chrome、Safari以及Opera等。

Selenium包括一系列的组件。

- **Selenium IDE：**是嵌入到Firefox浏览器的插件，用于在Firefox上录制与回放Selenium 脚本。图形化的界面可以形象地记录下用户在浏览器中的操作，非常方便使用者了解与学习。目前它只能在Firefox下使用，好在它能将录制好的脚本转换成各种Selenium WebDriver支持的程序语

言，进而扩展到更广泛的浏览器类型。

- **Selenium WebDriver:** 其实质上就是可以支持多种编程语言，并且有用于操作浏览器的一套API。支持多类型浏览器、跨操作系统平台（包括Linux、Windows以及Mac OS X），是真正意义上的跨浏览器测试工具。WebDriver为诸如Java、C#、Python、Ruby、PHP、JavaScript等语言分别提供了完备的、用于实现Web自动化测试的第三方库。
- **Selenium Standalone Server:** 包括被大家广泛了解的Selenium Grid、远程控制、分布式部署等，均可实现Selenium 脚本的高效执行与拓展。我们利用Grid使得自动化测试可以并行运行，甚至是在跨平台、异构的环境中运行，包括目前主流的移动端环境，如Android、iOS。

正如书名所述，这是一本介绍如何用Python语言调用Selenium WebDriver接口，进而实现对Web

应用自动化测试的指导书。本书描述了从Selenium安装配置到基本使用，再到创建、调试、运行自动化脚本等进阶的操作。当然在开始之前，你可能需要先具备一定的Python语言基础。

内容介绍

第1章基于Python的Selenium WebDriver入门
从安装Python、Selenium WebDriver开始，到我们如何选择适合的Python编辑器，以及我们小试牛刀的第一个自动化测试脚本，并且成功地将这一脚本运行在不同浏览器上。

第2章使用unittest编写单元测试 本章带领我们结合unittest实现单元测试。通过转换后的脚本，有助于我们更好地完善单元测试用例。借助unittest实现测试用例集的整体运行，并将HTML格式的测试结果及时推送给项目的相关人员。

第3章元素定位 本章告诉你如何通过浏览器自带的开发者模式去定位页面中各类型元素。Selenium通过获取这些元素的定位，进而实现模拟浏览器操作与参数捕获。这一章你将学会各种定位

元素的方法，包括XPath和CSS以及对应的示例。

第4章Selenium Python API介绍 学习如何通过WebDriver与包括页面元素、JavaScript 提示框、框架（frames）、窗口在内的各类对象进行交互，以及怎样进行浏览器回放、元素传值、鼠标点击、下拉菜单选择、多窗口切换等具体操作。

第5章元素等待机制 介绍多种设置等待方法，用于提高Selenium自动化测试脚本的稳定运行。带你理解显式等待或隐式等待的方法是如何应用于我们的测试脚本。

第6章跨浏览器测试 我们将深入学习如何在远程机器或Selenium Grid上通过Remote WebDriver实现测试脚本跨各类型浏览器的测试。Selenium Grid可使得我们在多浏览器与多操作系统的排列组合中兼容测试，甚至支持像PhantomJS这样的无UI界面的浏览器。本章的最后，我们还将了解Sauce

Labs 和BrowserStack等第三方外部测试服务（云测试）。

第7章移动端测试 我们使用Selenium WebDriver、Appium实现在包括iOS端、Android端以及Android模拟器在内的移动设备上的自动化测试。另外，本章还有App测试的具体示例。

第8章Page Object与数据驱动测试 介绍这两种重要的设计模式，引导我们搭建更持续、更高效的测试框架。其中，Page Object设计模式可帮助我们实现对界面细节的封装，并将一组用户行为构建在单个类中，提升自动化测试脚本的易读性和可复用性，从而达到更适应UI的频繁变化的目的。另外，我们还将学习用unittest实现数据驱动测试。

第9章Selenium WebDriver的高级特性 包括复杂的鼠标与键盘操作、cookies操作、窗口截屏，甚至录制整个测试过程。

第10章第三方工具与框架集成 通过Selenium与持续集成工具的搭配，我们可以轻松地搭建自动化验收测试框架。本章中展示了“通过Selenium创建自动化验收测试用例，然后细化基于UI的自动化测试脚本，最后配置持续集成工具Jenkins，最终实现了对被测程序每日构建、每日自动化验收测试的联动效果”的典型示例。

通过对本书的学习，你将能够用Python语言通过调用Selenium WebDriver接口，搭建属于你自己的Web应用自动化测试框架。

阅读前的准备工作

在阅读本书之前，你需要掌握Python语言基本语法以及Web前端的相关知识（如HTML、JavaScript、CSS和XML）。如果你能编写一些简单的包括循环、条件判断、定义类等语法的Python脚本，你就能轻松地理解本书中的示例代码。每行示例代码我们都花了很大精力去注释说明，就是希望你能达到最佳的学习效果。还有一些前期准备的软件、工具以及环境配置都在第1章有明确的说明，你需要在你的机器上准备好访问终端、Python解释器以及浏览器。

适合哪些人阅读

如果你从事QA或软件测试、软件开发、Web应用开发等相关工作，希望用Python语言调用Selenium WebDriver，以实现Web应用的自动化测试，那么这本书一定是你较好的选择！在学习Selenium理论之前，我们建议你掌握Python语言的基本语法。通过整本书的通篇学习，你将全面地理解Selenium WebDriver的相关知识，并且能有效地帮助你实现自动化测试。

约定

本书中，你可能会发现不同类型的信息，呈现出的文本风格不尽相同。我们在这里将罗列不同类型的文本风格以及对应的含义，方便你阅读。

代码文本的样式如下。

```
# create a new Firefox session
driver = webdriver.Firefox()
driver.implicitly_wait(30)
driver.maximize_window()
```

当我们想格外强调代码中的一部分时，相应的代码字体会被加粗，示例如下。

```
# run the suite
xmlrunner.XMLTestRunner(verbosity=2,output='test-reports').
run(smoke_tests)
```

命令行的输入\输出的样式如下。

```
pip install -U selenium
```

新的措辞与关键语句会被显示为粗体。关键语句就是指出现在系统界面、菜单项或对话框等位置的关键操作，例如，“在**Tools**下拉菜单中选择**Internet Options**”。



警告或重要的提示，会出现在这样的括号中。



提醒或小窍门，会出现在这样的括号中。

读者反馈

我们十分乐见读者的反馈，让我们了解你对本书的想法——包括好与不好的评价。因为你的反馈将使我们以后可以更好地为读者提供有价值的内容。

可以通过我们的邮箱地址 contact@epubit.com.cn 将你的反馈信息告诉我们，邮件的主题需要注明书籍的全名。

当然，如果你也有专注的主题，并且有兴趣编辑或撰写图书，欢迎你联系我们，邮箱为 zhangtao@ptpress.com.cn。

第1章 基于Python的Selenium WebDriver入门

Selenium可以自动地操纵浏览器来做很多事情，它可以模拟我们与浏览器的交互，比如，访问网站，单击链接，填写表单，提交表单，浏览网页等，而且支持大多数主流的浏览器。如果要使用Selenium WebDriver，我们首先要选择一种语言来编写自动化脚本，而这个编程语言需要有Selenium client library支持。

本书中，我们将使用支持Selenium client library的Python语言来编写自动化脚本。Python是一门被广泛应用的高级编程语言，它是非常容易上手的，而且它的语法使我们只需要简短的代码就可以用来表达思想。Python设计的初衷就非常强调代码的可

读性，它的基础架构使我们可以很方便地写无论大段的还是很少的程序代码，还提供大量的内置库、函数以及用户编写的第三方库，从而能够很容易地实现一些复杂的功能。

基于Python的Selenium WebDriver client library实现了所有Selenium WebDriver特性，而且能够通过Selenium standalone server来远程地和分布式地测试B/S项目。Selenium language bindings的开发者包含David Burns, Adam Goucher, Maik Röder, Jason Huggins, Luke Semerau, Miki Tebeka和Eric Allenin。

Selenium WebDriver client library 支持以下Python版本：2.6，2.7，3.2和3.3。

本章将介绍基于Python的Selenium WebDriver client library的安装步骤、基本特性和总体架构。

本章包括以下主题：

- 安装Python和Selenium包；
- 选择和设置Python编辑器；
- Selenium WebDriver基于Python编写实例脚本；
- 实现基于IE和Chrome的跨浏览器支持。

1.1 环境准备

作为学习使用基于Python的Selenium的第一步，我们需要在计算机上安装好需要的软件。在下面的章节中让我们一步步来配置所需的基础环境。

1.1.1 安装Python

在安装有Linux系统、Mac OS X系统和其他UNIX系统的计算机上，Python是系统默认安装好的。对于Windows系统，就需要另外单独安装Python了。基于不同平台的Python安装程序都可以在以下网站找到：<http://python.org/download/>。



本书所有的例子都是基于Python 2.7和Python 3.0编写，并在Windows 8 系统上经过测试的。

1.1.2 安装Selenium包

Selenium安装包里包含了Selenium WebDriver Python client library。为了使安装Selenium 包更简单，可以用pip安装工具：

<https://pip.pypa.io/en/latest/>。

使用pip，可以非常简单地通过下面的命令来安装和更新Selenium安装包。

```
pip install -U selenium
```

安装过程非常简单。该命令将会安装Selenium WebDriver client library在计算机上，包含我们使用Python来编写自动化脚本需要的所有模块和类。pip工具将会下载最新版本的Selenium安装包并安装在计算机上。这个可选的 -U 参数将会更新已经安装的旧版本至最新版。

也可以从网站下载最新版本的Selenium安装包：<https://pypi.python.org/pypi/selenium>。在页面的右上角单击下载按钮，下载后解压文件，然后通过下面的命令来安装。

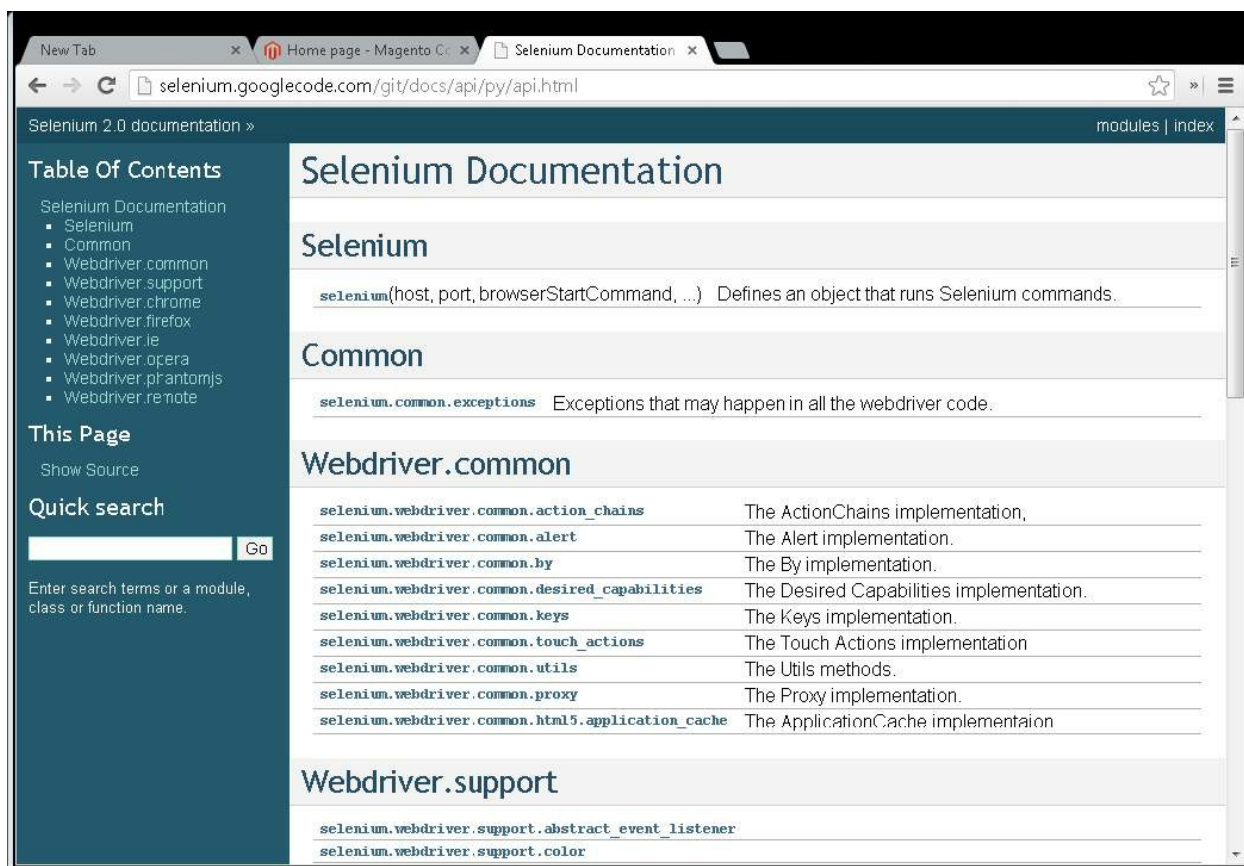
```
python setup.py install
```

1.1.3 浏览Selenium WebDriver Python文档

Selenium WebDriver Python client library 文档可以从以下网址查看。

<http://selenium.googlecode.com/git/docs/api/py/ap>

下面是截图。



这里提供了Selenium WebDriver的所有核心类和函数的详细信息。对于以下链接中的Selenium文档也要多加关注。

- 官方文档：<http://docs.seleniumhq.org/docs/>。这里有关于Selenium所有组件的说明文档以及基于一些所支持的语言编写的实例。
- Selenium Wiki地址：

<https://code.google.com/p/selenium/w/list>。这里列举了将在后面的章节能够看到的有用的主题。

1.1.4 选择一个**IDE**

现在已经安装好了Python和Selenium WebDriver，还需要一个代码编辑器（IDE）来编写自动化脚本。一个好的**IDE**能够帮助我们提高产出，而且还能做一些其他的事情让编码变得简单。当我们用简单的编辑器来编写Python代码，比如Emacs、Vim或Notepad，用编辑器会让事情变得简单多了。其实有很多的IDE可供选择。一般来说，一款好的IDE能够通过以下一些特性帮忙开发者提高开发速度并节省编码时间：

- 图形化界面编辑器，具备代码编译和代码自动补全功能；
- 方便地查看类和函数代码；
- 语法高亮显示；

- 具备项目管理功能；
- 支持代码模板；
- 具备支持单元测试和调试的工具；
- 支持源码管理。

如果你是个Python开发新手，或者是第一次接触Python的测试工程师，你的研发同事们能够帮助你安装和配置相应的IDE。

然而，如果你是第一次接触Python而不知道选择哪个IDE，这里有些建议可以帮助你做出选择。

1.1.4.1 PyCharm

PyCharm是JetBrains公司出品的软件，该公司是专业的软件开发工具的引领者，产品包含大家熟知的IntelliJ IDEA、RubyMine、PhpStorm和TeamCity。

PyCharm是一款设计精巧、功能强大、应用广

泛而且工作良好的IDE。它继承了JetBrains 公司其他产品的一贯经验，拥有很多能够提升软件开发效率的特性。

PyCharm支持Windows系统、Linux系统和Mac系统。要想知道更多PyCharm的特性，可以访问以下网址：<http://www.jetbrains.com/pycharm/>。

PyCharm有两种版本——社区版和专业版。社区版是免费的，而专业版是需要付费的。下面是PyCharm社区版运行一个简单Selenium脚本例子的截图。

社区版能够很好地构建和运行Selenium脚本，并提供调试支持。在本书后面的章节将会使用PyCharm。本章后面的部分，我们一步步来安装PyCharm，并用它来创建第一个Selenium脚本。



本书中所有的例子都是用PyCharm构建的，不过读者也可以很容易地应用别的IDE来构建这些例子。

```
setests_final - [C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final] - ...\searchproducts_with_ie.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help

setests_final > searchproducts_with_ie.py >
Project searchproducts.py x searchproducts_with_ie.py x

setests_final (C:\Users\amitr\Desktop\Mrunmayee)
  IEDriverServer.exe
  searchproducts.py
  searchproducts_with_ie.py
  External Libraries

import os
from selenium import webdriver

# get the path of IEDriverServer
dir = os.getcwd()
ie_driver_path = dir + "\IEDriverServer.exe"

# create a new Internet Explorer session
driver = webdriver.Ie(ie_driver_path)
driver.implicitly_wait(30)
driver.maximize_window()

# navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

# get the search textbox
search_field = driver.find_element_by_name("q")
search_field.clear()

# enter search keyword and submit
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names displayed
# currently on result page using find_elements_by_xpath method
products = driver.find_elements_by_xpath("//h2[@class='product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products:"

# iterate through each anchor element and
# print the text i.e. name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()

Run searchproducts_with_ie
C:\Python27\python.exe C:/Users/amitr/Desktop/Mrunmayee/Final/setests_final/searchproducts_with_ie.py
Found 2 products:
Madison Earbuds
Madison Overear Headphones
Process finished with exit code 0
```

1.1.4.2 PyDev Eclipse plugin

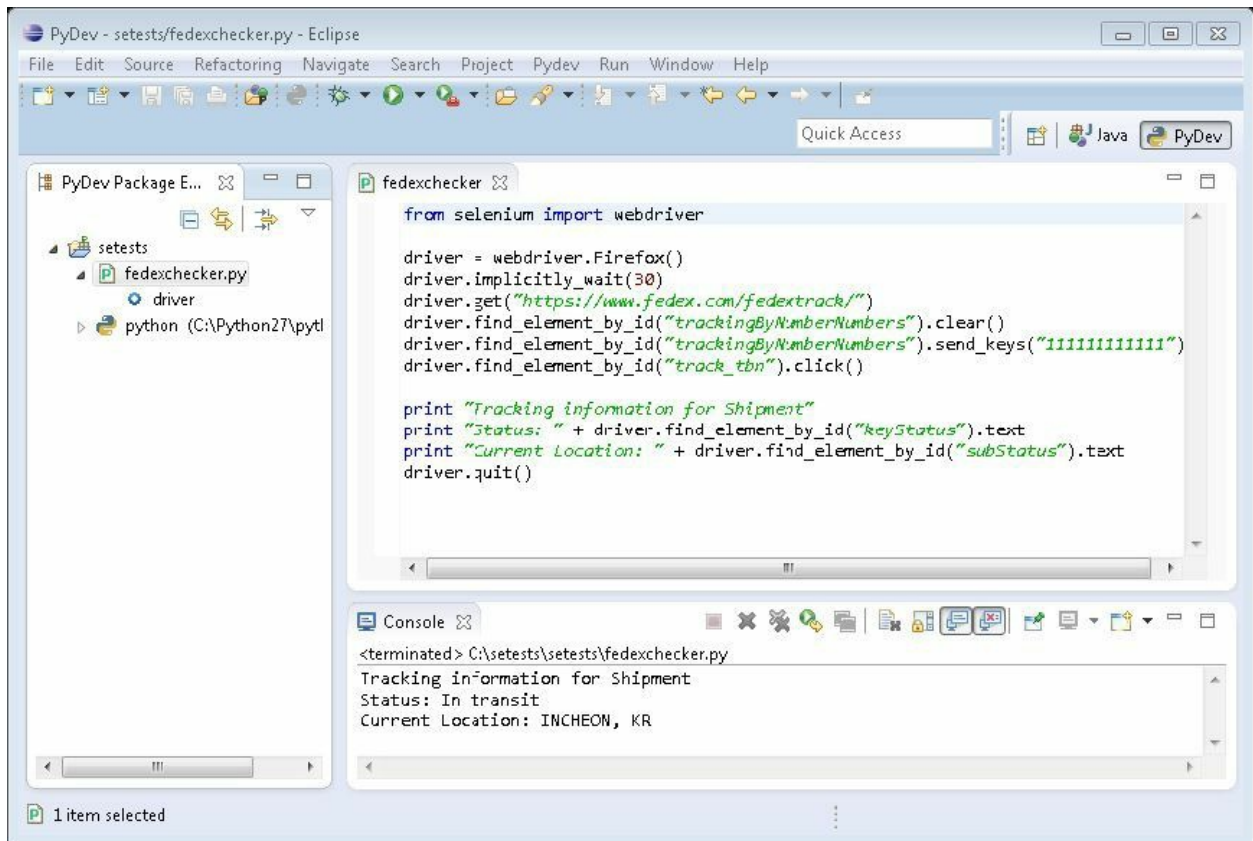
PyDev Eclipse plugin是另一款在Python开发者中应用广泛的代码编辑器。Eclipse是一款知名的开

源代码编辑器，主要应用于构建Java程序，它通过插件式架构设计从而实现对其他多种编程语言的支持。

Eclipse是一款跨平台的IDE，支持Windows系统、Linux系统和Mac系统。可以在以下网址获取Eclipse的最新版
本：<http://www.eclipse.org/downloads>。

PyDev plugin需要在安装完Eclipse后单独安装。可以按照Lars Vogel编写的安装指南来安装PyDev：<http://www.vogella.com/tutorials/Python/article.html>
也可以在以下网址查看安装说明：
<http://pydev.org/>。

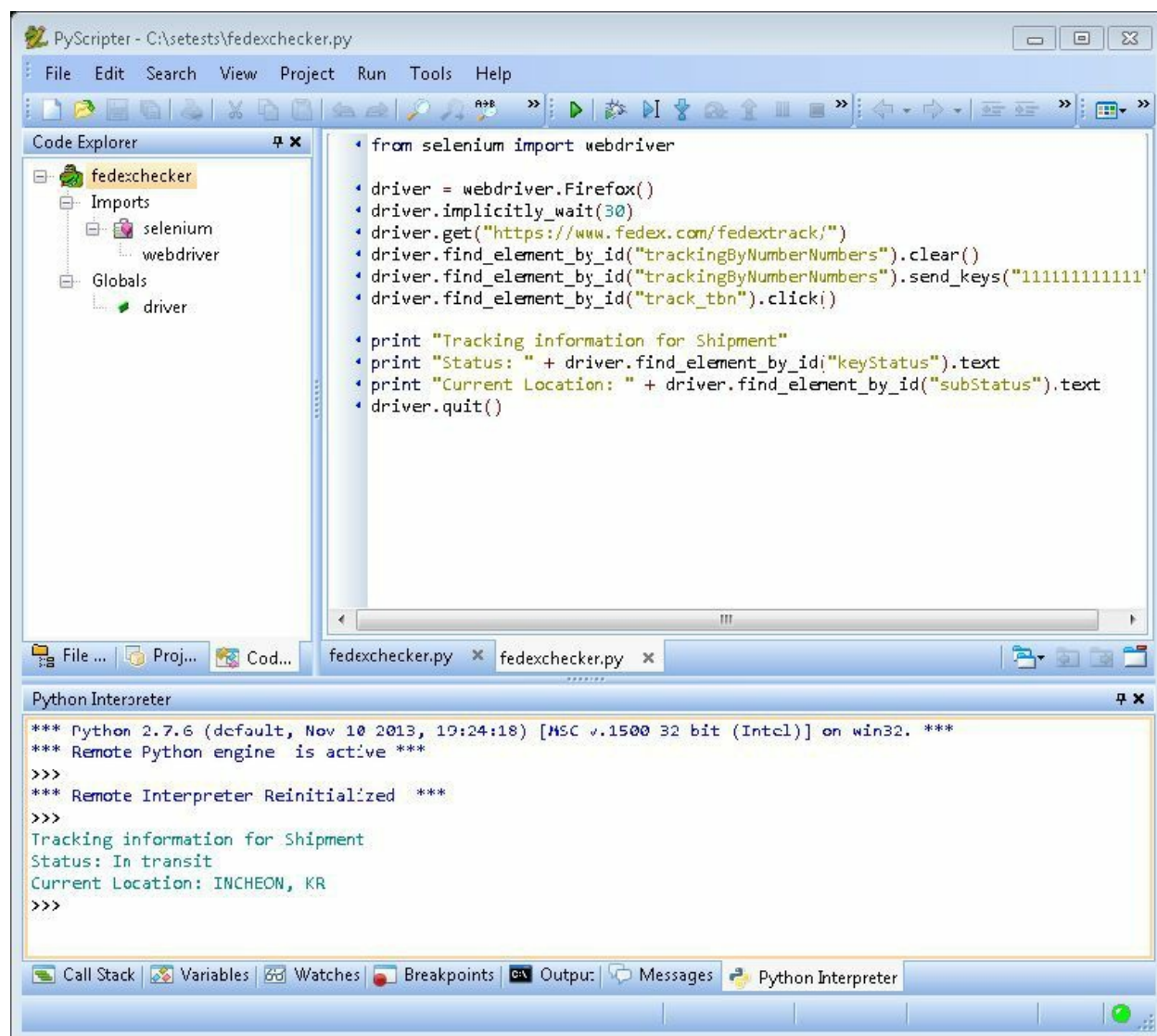
下面是使用PyDev Eclipse plugin 运行一个简单Selenium脚本例子的截图。



1.1.4.3 PyScripter

对于Windows系统的用户，PyScripter也是一个很好的选择。它是一款开源的、轻量级的IDE。PyScripter像其他流行的IDE一样具有代码编译和代码自动补全的特性，并且提供测试和调试功能支持。可以从以下网站下载该软件并获取更多的信息：<https://code.google.com/p/pyscripter/>。

下面是在 PyScripter 上运行一个简单 Selenium 脚本实例的截图。



1.1.5 PyCharm 设置


我们看过这些可选择的IDE后，回到PyCharm

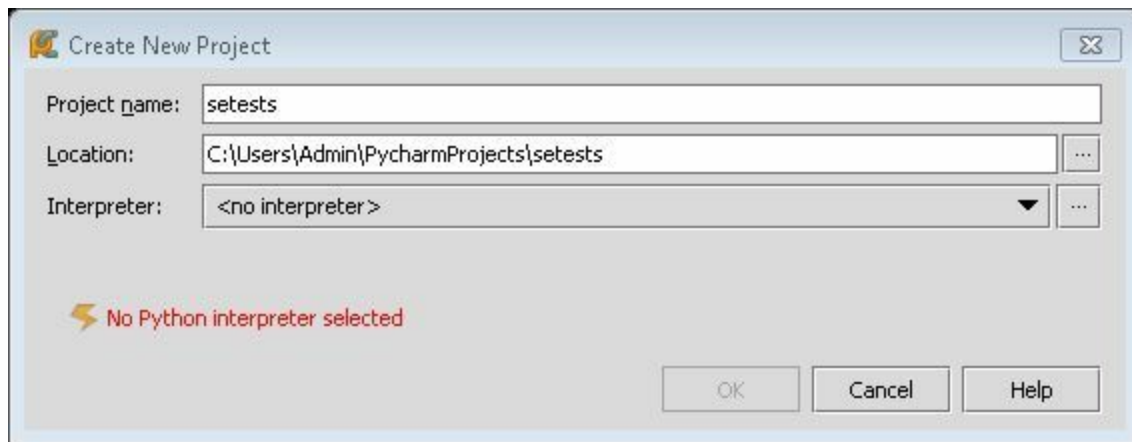
的设置。本书中所有的例子脚本都是通过PyCharm创建的，不过读者也可以使用其他IDE来创建并运行这些实例。下面将通过以下步骤来设置PyCharm，开始我们的Selenium Python之旅。

(1) 从JetBrains官网下载和安装PyCharm。

(2) 启动PyCharm社区版，在启动页面上单击**Create New Project**选项。



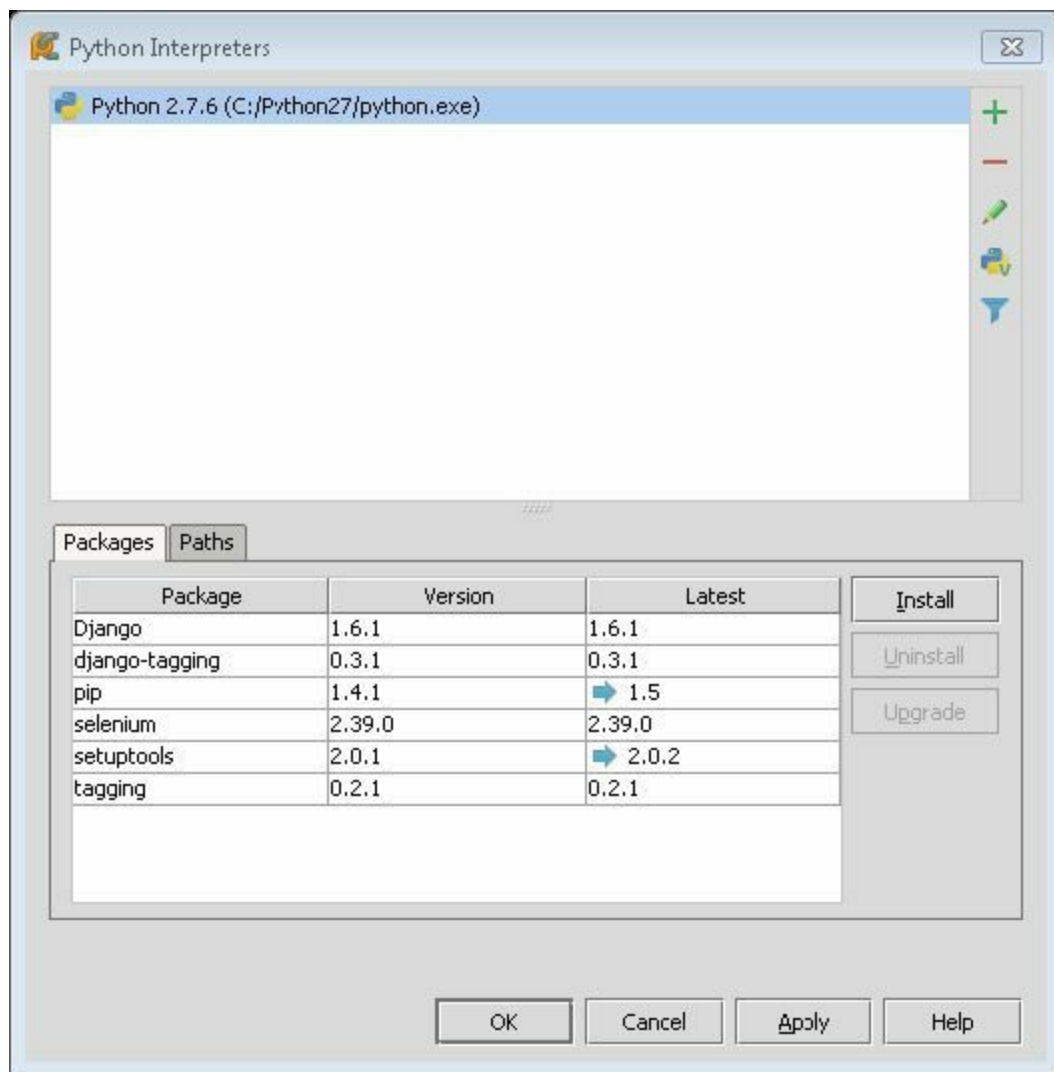
(3) 在Create New Project对话框，参考下面的截图，在Project name文本框内输入工程名称。在本例中，setests作为工程名称。第一次运行PyCharm还需要配置解释器。单击Interpreter右侧的按钮来配置解释器。



(4) 在弹出的**Python Interpreter**对话框中，单击加号，PyCharm会显示出已经安装好的解释器路径。从Select Interpreter Path选择对应的解释器。

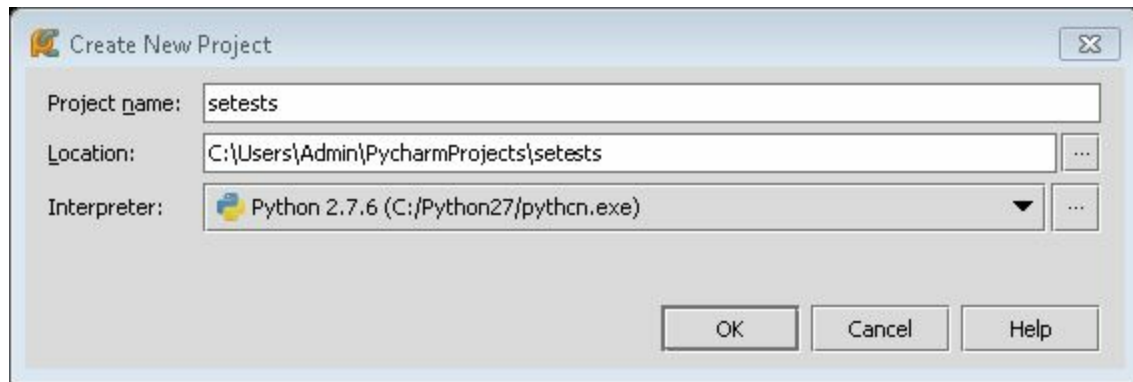


(5) PyCharm 将会配置好刚才选择的解释器。在Packages选项卡会显示Python安装包内自带的一些工具包。单击**Apply**按钮，然后单击**OK**按钮。



(6) 返回到**Create New Project**对话框，单击

OK按钮，项目创建成功。



1.2 第一个Selenium Python脚本

我们现在可以开始创建和运行自动化测试脚本了。就从Selenium WebDriver开始，然后创建一个Python脚本，用Selenium WebDriver提供的类和方法模拟用户与浏览器的交互。

我们会使用一个简单的Web应用程序（本书上大多数例子都是基于这个应用程序）。这个简单的Web应用程序是基于一个著名电子商务框架**Magento**构建的。你可以在以下网址里找到这个应用程序：<http://demo.magentocommerce.com/>。



示例代码下载

如果你是在<http://packtpub.com>购买本书，你可以通过你的账号在该网址上下载示

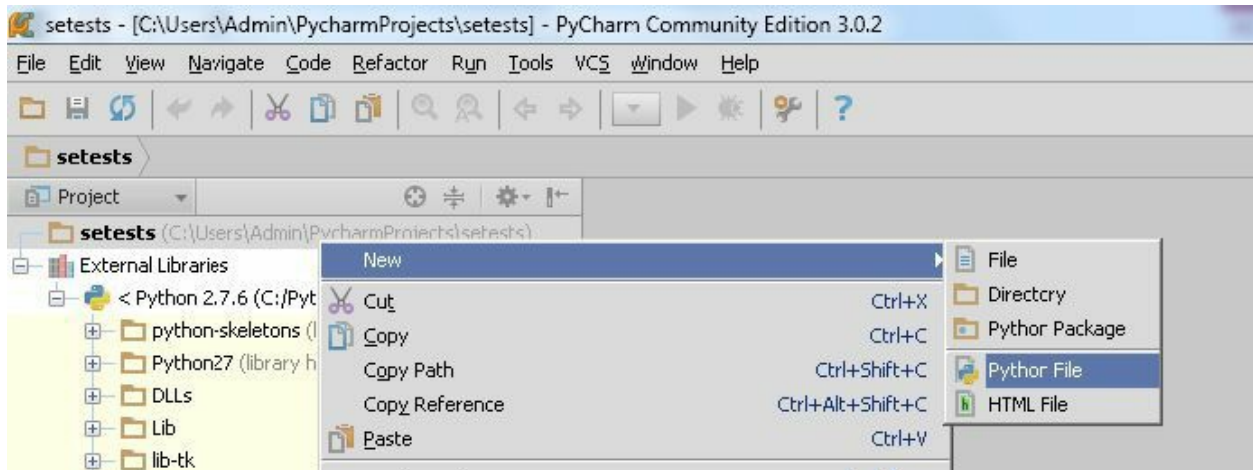
例代码文件。如果你是从其他地方购买本书，你可以访问
<http://www.packtpub.com/support>并注册，我们会把示例代码文件直接发送到你的邮箱中。

示例代码也被托管在github中，访问地址为：

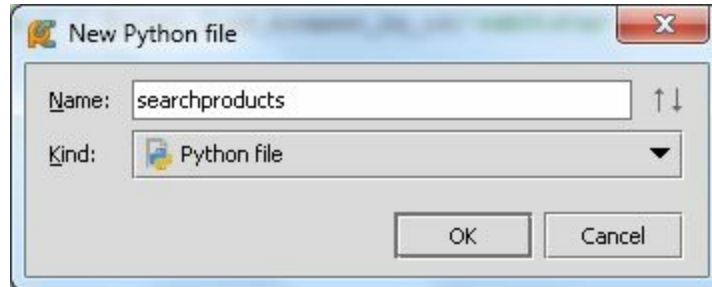
<https://github.com/upgundecha/learnsewithpython>。

在这个简单的脚本中，我们会通过接下来的步骤去访问这个应用程序，搜索产品并在搜索结果页面中列出产品的名称。

（1）我们使用早前在部署PyCharm环境时创建的项目。创建一个引用了Selenium WebDriver client library的Python脚本。在项目资源管理器视图中，右击setests，在弹出的菜单中依次选择 **New → Python File**。



(2) 在**New Python file**对话框中，在文件名文本框中输入“searchproducts”，然后单击OK按钮。



(3) PyCharm会在代码编写区域增加一个名为searchproducts.py的新页签。复制下列代码到searchproducts.py中。

```
from selenium import webdriver  
  
# create a new Firefox session
```

```
driver = webdriver.Firefox()
driver.implicitly_wait(30)
driver.maximize_window()

# navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

# get the search textbox
search_field = driver.find_element_by_name("q")
search_field.clear()

# enter search keyword and submit
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names
displayed
# currently on result page using find_elements_by_xpath
method
products = driver.find_elements_by_xpath("//h2[@class='
product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products:"

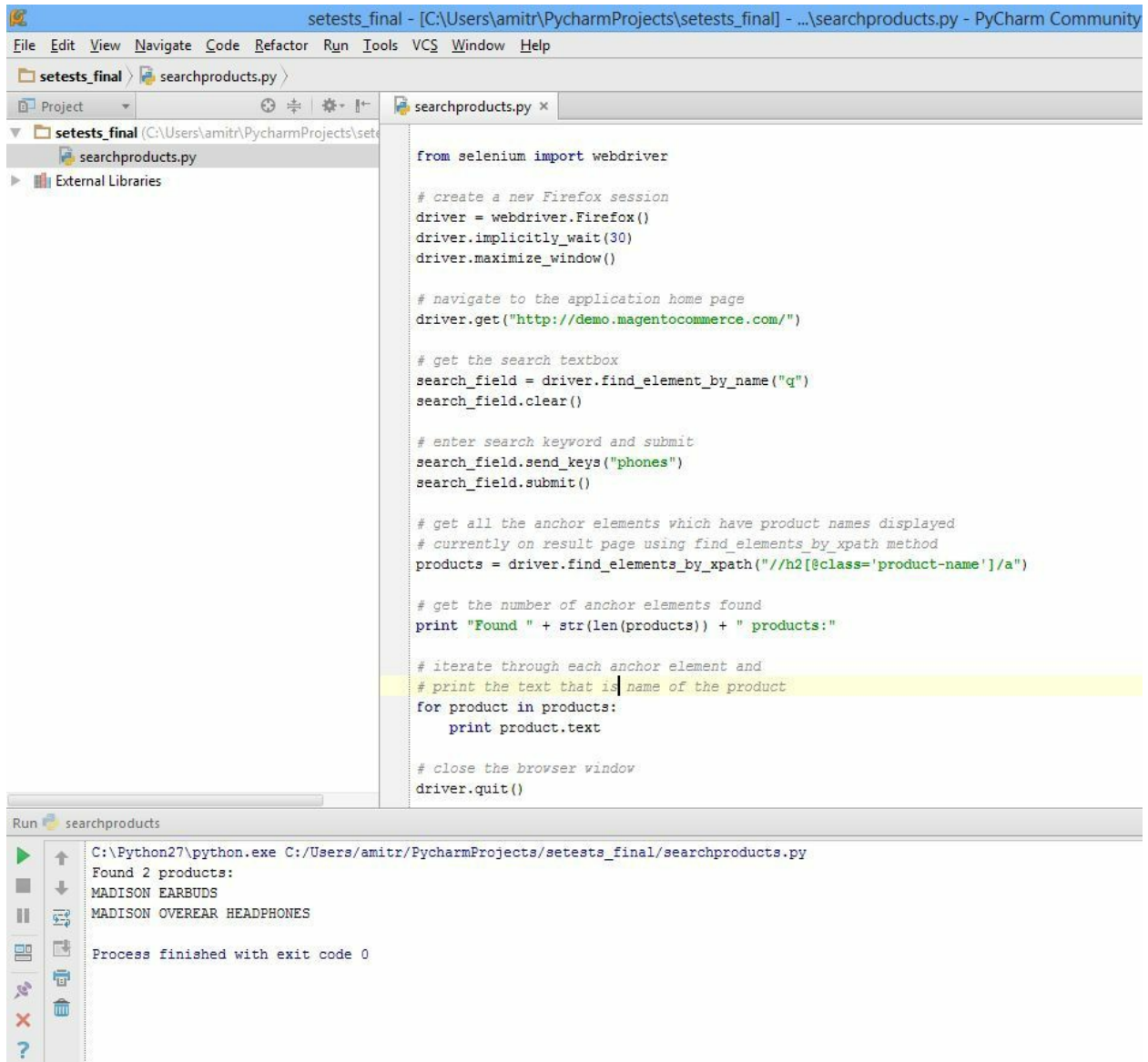
# iterate through each anchor element and print the text
that is
# name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()
```



如果你使用的是其他IDE编译工具，请同样创建一个新的文件，复制代码到文件中并保存为searchproducts.py。

(4) 可以通过以下方式运行脚本：在PyCharm代码窗口中使用快捷键Ctrl + Shift+ F10或者在**Run**菜单中选择**Run 'searchproducts'**命令。脚本开始执行，你会看到新弹出一个Firefox浏览器窗口访问演示网址，接着在Firefox浏览器窗口中会看到被执行的Selenium命令。如果一切运行顺利，最后脚本会关闭Firefox浏览器窗口。如下图所示，这个脚本会在PyCharm的控制台中打印产品的清单。



The screenshot displays the PyCharm IDE interface. The top toolbar includes menus like File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, and Help. The left sidebar shows the project structure for 'setests_final' with a file named 'searchproducts.py'. The main editor window contains the following Python code:

```
from selenium import webdriver

# create a new Firefox session
driver = webdriver.Firefox()
driver.implicitly_wait(30)
driver.maximize_window()

# navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

# get the search textbox
search_field = driver.find_element_by_name("q")
search_field.clear()

# enter search keyword and submit
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names displayed
# currently on result page using find_elements_by_xpath method
products = driver.find_elements_by_xpath("//h2[@class='product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products:"

# iterate through each anchor element and
# print the text that is name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()
```

Below the editor, the 'Run' console shows the execution of 'searchproducts.py' using 'C:\Python27\python.exe'. The output is:

```
Found 2 products:
MADISON EARBUDS
MADISON OVEREAR HEADPHONES
Process finished with exit code 0
```



我们也可以在命令行中运行脚本。打开命令行工具，切换到setests项目所在的目录中，运行以下命令：

```
python searchproducts.py
```

在本书后面部分，我们更喜欢选择使用命令行方式去执行测试脚本。

接下来，我们将会花点时间分析刚才创建的脚本。我们分析每个语句，初步地认识Selenium WebDriver。在本书的后面部分还有很多这样的分析。

Selenium.webdriver模块实现了Selenium所支持的各种浏览器驱动程序类，包括Firefox浏览器、Chrome浏览器、IE浏览器、Safari浏览器和多种其他浏览器。另外，RemoteWebDriver则是用于调用远程机器进行浏览器测试的。

我们需要从Selenium包中导入WebDriver才能使用Selenium WebDriver方法。

```
from selenium import webdriver
```

接着，我们还需要选用一个浏览器驱动实例，它会提供一个接口去调用Selenium命令来跟浏览器交互。在这个例子中，我们使用的是Firefox浏览器。我们可以通过下方命令来创建一个Firefox浏览器驱动实例。

```
driver = webdriver.Firefox()
```

在运行期间，这会加载一个新的Firefox浏览器窗口。我们也可以在这个驱动上设置一些参数，如：

```
driver.implicitly_wait(30)  
driver.maximize_window()
```

我们使用30秒隐式等待时间来定义Selenium执行步骤的超时时间，并且调用Selenium API来最大化浏览器窗口。我们会在第5章“元素等待机制”中学习更多关于隐式等待的内容。

接着，我们使用示例程序的URL作为参数，通过调用`driver.get()`方法访问该应用程序。在`get()`方法被调用后，`WebDriver`会等待，一直到页面加载完成才继续控制脚本。

在加载页面后，`Selenium`会像用户真实使用那样，和页面上各种各样的元素交互。例如，在应用程序的主页，我们需要在输入框中输入一个搜索内容，然后单击**Search**按钮。这些元素作为HTML输入元素实现，`Selenium`需要找到这些元素来模拟用户操作。`Selenium WebDriver`提供多种方法来定位和操作这些元素，例如设置值，单击按钮，在下拉组件中选择选项等。我们可以在第3章“元素定位”中了解更多。

在这个例子中，我们使用`find_element_by_name`方法来定位搜索输入框。这个方法会返回第一个`name`属性值与输入参数匹配的

元素。HTML元素是用标签和属性来定义的，我们可以使用这些信息来定位一个元素，步骤如下。

(1) 在这个例子中，搜索输入框有一个值为q的name属性，我们使用这个属性来定位，代码如下。

```
search_field = driver.find_element_by_name("q")
```

(2) 一旦找到这个搜索输入框，我们可以使用clear()方法来清理之前的值（如果搜索输入框已经有值的话），并且通过send_keys()方法输入新的特定的值。接着我们通过调用submit()方法提交搜索请求。

```
search_field.clear()  
search_field.send_keys("phones")  
search_field.submit()
```

(3) 在提交搜索请求后，Firefox浏览器会加载结果页面。结果页面中有一系列与搜索项

（phones）匹配的产品。我们可以读取结果列表，并且可以使用find_elements_by_xpath方法获取路径是以<a>标签结尾的所有产品名称。它将会返回多于1个的元素列表。

```
products =  
    driver.find_elements_by_xpath("//h2[@class=  
        'product-name']/a")
```

（4）接着，我们打印在页面中展示的产品个数（即符合路径以<a>标签结尾的元素个数）和产品的名称（即<a>标签的text属性值）。

```
print "Found " + str(len(products)) + " products:"  
  
for product in products:  
    print product.text
```

（5）在脚本的最后，我们使用driver.quit()方法来关闭Firefox浏览器。

```
driver.quit()
```

这个例子直观地向我们展示如何使用Selenium WebDriver和Python配合来创建一个简单的自动化脚本。我们在这个脚本里面并没有测试什么。在本书后面的章节，我们将会扩展这个简单的脚本为一组测试脚本，并且会引用多个其他库和Python的功能。

1.3 支持跨浏览器

目前我们已经在Firefox浏览器构建并运行了脚本。Selenium支持各种浏览器，读者可以在不同的浏览器中进行自动化测试。它支持的浏览器包括IE浏览器、Google Chrome浏览器、Safari浏览器、Opera浏览器，甚至是像PhantomJS这样的无UI界面的浏览器。接下来的部分，我们会修改刚才创建的脚本，以便在IE浏览器和Google Chrome浏览器中运行脚本，以此来验证Selenium WebDriver跨浏览器的兼容性。

1.3.1 设置IE浏览器

在IE浏览器中运行脚本步骤会多一些。我们需要下载并安装InternetExplorerDriver。InternetExplorerDriver是一个独立的可执行的服务，它实现WebDriver的协议，使得WebDriver可以与测

试脚本和IE浏览器交互。InternetExplorerDriver支持Windows XP、Vista、Windows 7和Windows 8操作系统下的主要IE版本。通过以下步骤安装InternetExplorerDriver。

(1) 在<http://www.seleniumhq.org/download/>中下载InternetExplorerDriver服务。你可以根据自己的操作系统来选择下载32位或64位版本。

(2) 在下载完成后，解压文件，并把文件复制到存储脚本的目录中。

(3) 在IE 7及其以上版本，每个区域的保护模式设置一定要有相同的值。在每个区域中保护模式要么启用，要么关闭。设置保护模式的步骤如下。

① 在工具菜单下选择Internet选项。

② 在**Internet**选项对话框中，单击安全标签页。

③ 在“选择区域以查看或更改安全设置”中选择每一个区域，确定每个区域的保护模式的值保持一致（要么选中，要么不选中）。所有区域用相同的设置，如下图所示。





在使用InternetExplorerDriver时，注意保持浏览器缩放等级设置成100%，以此来保证鼠标的单击事件能点到正确的坐标。

(4) 修改脚本使其支持IE浏览器。我们通过以下方式使用IE替代Firefox实例。

```
import os
from selenium import webdriver

# get the path of IEDriverServer
dir = os.path.dirname(__file__)
ie_driver_path = dir + "\IEDriverServer.exe"

# create a new Internet Explorer session
driver = webdriver.Ie(ie_driver_path)
driver.implicitly_wait(30)
driver.maximize_window()

# navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

# get the search textbox
search_field = driver.find_element_by_name("q")
search_field.clear()
```

```
# enter search keyword and submit
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names
displayed
# currently on result page using find_elements_by_xpath
method
products = driver.find_elements_by_xpath("//h2[@class='
product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products:"

# iterate through each anchor element and print the tex
t that is
# name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()
```

在这个脚本中，在创建IE浏览器实例时，我们传递了InternetExplorerDriver的路径。

（5）运行脚本后，Selenium会加载InternetExplorerDriver服务，用它来启动浏览器和执行脚本。InternetExplorerDriver服务在Selenium脚本

和浏览器之间扮演类似中介角色。实际执行的步骤与我们在Firefox浏览器观察的类似。



在<https://code.google.com/p/selenium/wiki/>

InternetExplorerDriver中可以获取更多关于IE的重要设置。

在<https://code.google.com/p/selenium/wiki/>

DesiredCapabilities 查阅

DesiredCapabilities的文章。

1.3.2 设置Google Chrome浏览器

在Google Chrome浏览器中设置和运行脚本的步骤与IE浏览器的相似。我们需要下载ChromeDriver服务。ChromeDriver服务是一个由Chromium team开发维护的独立的服务，它支持

Windows操作系统、Linux操作系统和Mac操作系统。使用以下步骤来设置ChromeDriver服务。

(1) 在
<http://chromedriver.storage.googleapis.com/index.html>
下载ChromeDriver服务。

(2) 下载完ChromeDriver服务后，解压文件，并把文件复制到存储脚本的目录中。

(3) 修改脚本使其支持Chrome浏览器。我们通过以下方式创建Chrome实例，用此来替换Firefox浏览器实例。

```
import os
from selenium import webdriver

# get the path of chromedriver
dir = os.path.dirname(__file__)
chrome_driver_path = dir + "\chromedriver.exe"
# remove the .exe extension on linux or mac platform

# create a new Chrome session
driver = webdriver.Chrome(chrome_driver_path)
```

```
driver.implicitly_wait(30)
driver.maximize_window()

# navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

# get the search textbox
search_field = driver.find_element_by_name("q")
search_field.clear()

# enter search keyword and submit
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names
displayed
# currently on result page using find_elements_by_xpath
method
products = driver.find_elements_by_xpath("//h2[@class='
product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products: "

# iterate through each anchor element and print the tex
t that is
# name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()
```

在这个脚本中，在创建Chrome浏览器实例时，

我们传递了ChromeDriver的路径。

（4）运行脚本后，Selenium会加载ChromeDriver服务，用它来启动浏览器和执行脚本。

实际执行的步骤与我们在Firefox浏览器观察的类似。



想了解更多关于ChromeDriver，请访问
<https://code.google.com/p/selenium/wiki/ChromeDriver>和
<https://sites.google.com/a/chromium.org/chrome-driver/home>。

1.4 章节回顾

在本章中，我们介绍了Selenium和它的组件。通过pip工具安装了Selenium包。接着介绍了多个用于编写Selenium和Python代码的编辑器和IDE工具，部署了PyCharm环境。然后我们通过创建基于示例程序的测试脚本，成功运行在Firefox浏览器，并分析了整个过程。最后，我们举一反三（分别在IE浏览器和Chrome浏览器中配置和运行脚本）来验证Selenium WebDriver的跨浏览器的特性。

在下一章，我们将学习如何通过Selenium WebDriver使用unittest库来创建自动化单元测试。我们也将学习如何创建并运行一组测试脚本。

第2章 使用unittest编写单元测试

Selenium WebDriver是一个浏览器自动化测试的API集合。它提供了很多与浏览器自动化交互的特性，并且这些API主要是用于测试Web程序。如果仅仅使用Selenium WebDriver，我们无法实现执行测试前置条件、测试后置条件，比对预期结果和实际结果，检查程序的状态，生成测试报告，创建数据驱动的测试等功能。在本章，我们将学习如何使用unittest来创建基于Python的Selenium WebDriver测试脚本。

本章包含以下主题：

- 什么是unittest？
- 使用unittest来写Selenium WebDriver测试；

- 用TestCase类来实现一个测试;
- 学习unittest提供的不同类型的assert方法;
- 为一组测试创建TestSuite;
- 使用unittest扩展来生成HTML格式的测试报告。

2.1 unittest单元测试框架

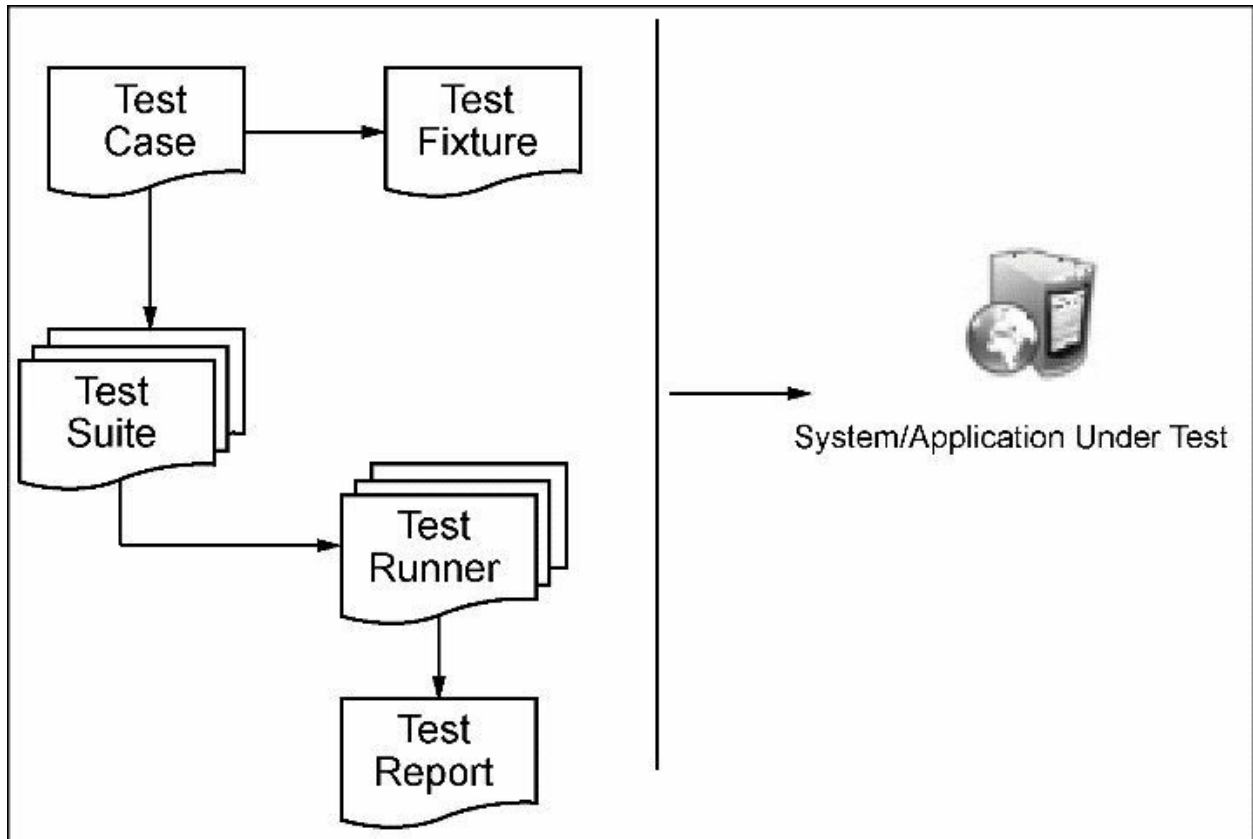
unittest（一般称为PyUnit）是从Java程序开发中广泛应用的JUnit启发而来的。我们可以使用unittest为任何项目创建全面的测试套件。unittest也是Python中用来测试各种标准类库模块的，甚至包括unittest自己。可以在以下网址查看unittest的文档：<http://docs.python.org/2/library/unittest.html>。

unittest使我们具备创建测试用例、测试套件、测试夹具的能力。可以通过下面的图来了解所有的组件。

- **Test Fixture**（测试夹具）：通过使用测试夹具，可以定义在单个或多个测试执行之前的准备工作和测试执行之后的清理工作。
- **Test Case**（测试用例）：一个测试用例是在unittest中执行测试的最小单元。它通过unittest

提供的assert方法来验证一组特定的操作和输入以后得到的具体响应。unittest提供了一个名称为TestCase的基础类，可以用来创建测试用例。

- **Test Suite**（测试套件）：一个测试套件是多个测试或测试用例的集合，是针对被测程序的对应的功能和模块创建的一组测试，一个测试套件内的测试用例将一起执行。
- **Test Runner**（测试执行器）：测试执行器负责测试执行调度并且生成测试结果给用户。测试执行器可以使用图形界面、文本界面或者特定的返回值来展示测试执行结果。
- **Test Report**（测试报告）：测试报告用来展示所有执行用例的成功或者失败状态的汇总，执行失败的测试步骤的预期结果与实际结果，还有整体运行状况和运行时间的汇总。



通过与unittest类似的xUnite测试框架创建的测试被拆分为3部分，即3A's，具体如下。

- **Arrange:** 是用来初始化测试的前置条件，包含初始化被测试的对象、相关配置和依赖。
- **Act:** 用来执行功能操作。
- **Assert:** 用来校验实际结果与预期结果是否一致。

我们在本章接下来的内容中将应用此方法来使用unittest创建测试。



我们将在本书接下来的部分使用unittest来创建和运行基于Selenium WebDriver的测试。另外，Python还有些具备额外特性的其他测试框架，例如：



- **Nose:** 此框架扩展了unittest并且提供了自动搜索和运行测试的功能，也提供了一些插件来创建高级的测试。可以在以下网站查看关于Nose的更多信息。
<https://nose.readthedocs.org/en/latest/>
- **Pytest:** Pytest是另外一个测试框架，它提供了一些基于Python来编写和运行单元

测试的高级特性。可以在以下网站查看关于Pytest的更多信息。

<http://pytest.org/latest/>

2.1.1 TestCase类

我们可以通过继承TestCase类并且在测试类中为每一个测试添加测试方法来创建单个测试或者一组测试。为了创建测试，我们需要使用TestCase类中的assert或者使用其中的一种assert方法。每个测试最重要的任务是调用assertEqual()来校验预期结果，调用assertTrue()来验证条件，或者调用assertRaises()来验证预期的异常。

除了添加测试，我们可以添加测试夹具——setUp()方法和tearDown()方法，创建或处置测试用例所需要的任何对象和条件。

让我们开始使用unittest，首先通过继承

TestCase类然后添加一个测试方法，来为第1章（基于Python的Selenium WebDriver入门）中的例子脚本写一个简单的测试。

我们需要先引入unittest模块，然后定义一个继承于TestCase 类的子类，具体如下。

```
import unittest
from selenium import webdriver

class SearchTest (unittest.TestCase):
```

2.1.1.1 setUp()方法

一个测试用例是从setUp()方法开始执行的，我们可以用这个方法在每个测试开始前去执行一些初始化的任务。可以是这样的初始化准备：比如创建浏览器实例，访问URL，加载测试数据和打开日志文件等。

此方法没有参数，而且不返回任何值。当定义了一个setUp()方法，测试执行器在每次执行测试方

法之前优先执行该方法。在下面的例子里，我们将用setUp()方法来创建Firefox的实例，设置properties，而且在测试开始执行之前访问到被测程序的主页。例子如下。

```
import unittest
from selenium import webdriver

class SearchTests(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com/")
```

2.1.1.2 编写测试

有了setUp()方法，现在可以写一些测试用来验证我们想要测试的程序的功能。在这个例子里，我们将搜索一个产品，然后检查是否返回一些相应的结果。与setUp()方法相似，test方法也是在TestCase

类中实现。重要的一点是我们需要给测试方法命名为test开头。这种命名约定通知test runner哪个方法代表测试方法。

对于test runner能找到的每个测试方法，都会在执行测试方法之前先执行setUp()方法。这样做有助于确保每个测试方法都能够依赖相同的环境，无论类中有多少测试方法。我们将使用简单的assertEqual()方法来验证用程序搜索该术语返回的结果是否和预期结果相匹配。我们将在本章后面内容探讨更多关于断言的内容。

添加一个新的测试方法

test_search_by_category(), 通过分类来搜索产品，然后校验返回的产品的数量是否正确，具体如下。

```
import unittest
from selenium import webdriver

class SearchTests(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
```

```

self.driver = webdriver.Firefox()
self.driver.implicitly_wait(30)
self.driver.maximize_window()

# navigate to the application home page
self.driver.get("http://demo.magentocommerce.com/")

def test_search_by_category(self):
    # get the search textbox
    self.search_field = self.driver.find_element_by_name("q")
    self.search_field.clear()

    # enter search keyword and submit
    self.search_field.send_keys("phones")
    self.search_field.submit()

    # get all the anchor elements which have product names
    # displayed currently on result page using
    # find_elements_by_xpath method
    products = self.driver.find_elements_by_xpath(
        ("//h2[@class='product-name']/a")
    self.assertEqual(2, len(products))

```

2.1.1.3 代码清理

类似于setUp()方法在每个测试方法之前被调用，TestCase类也会在测试执行完成之后调用tearDown()方法来清理所有的初始化值。一旦测试

被执行，在setUp()方法中定义的值将不再需要，所以最好的做法是在测试执行完成的时候清理掉由setUp()方法初始化的数值。在我们的例子里，在测试执行完成后，就不再需要Firefox的实例。我们将在tearDown()方法中关闭Firefox实例，如下代码所示。

```
import unittest
from selenium import webdriver

class SearchTests(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com/")

    def test_search_by_category(self):
        # get the search textbox
        self.search_field = self.driver.find_element_by_name("q")
        self.search_field.clear()

        # enter search keyword and submit
        self.search_field.send_keys("phones")
```

```
        self.search_field.submit()

        # get all the anchor elements which have product names
        # displayed currently on result page using
        # find_elements_by_xpath method
        products = self.driver.find_elements_by_xpath(
            ("//h2[@class='product-name']/a")
        )
        self.assertEqual(2, len(products))

    def tearDown(self):
        # close the browser window
        self.driver.quit()
```

2.1.1.4 运行测试

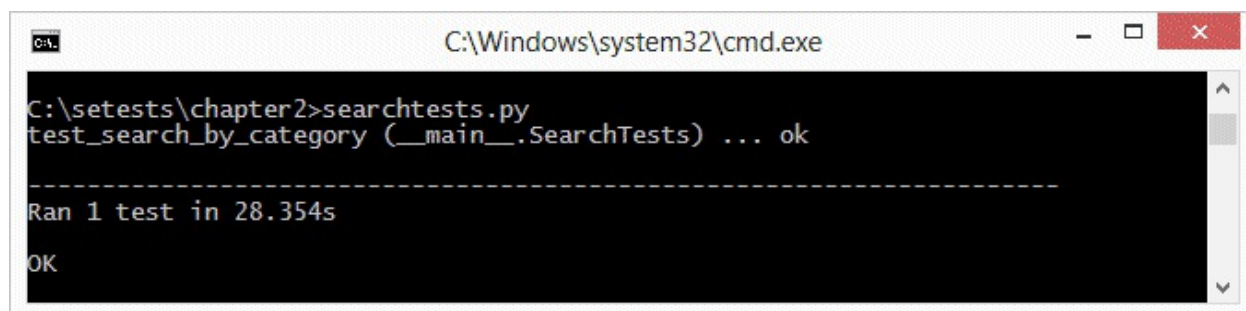
为了通过命令行运行测试，我们可以在测试用例中添加对main方法的调用。我们将传递verbosity参数以便使详细的测试总量展示在控制台。

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

我们可以把测试脚本保存为普通的Python脚本。在这个例子中，把测试保存为searchtests.py。保存文件以后，我们可以通过下面的命令行来执行该测试。

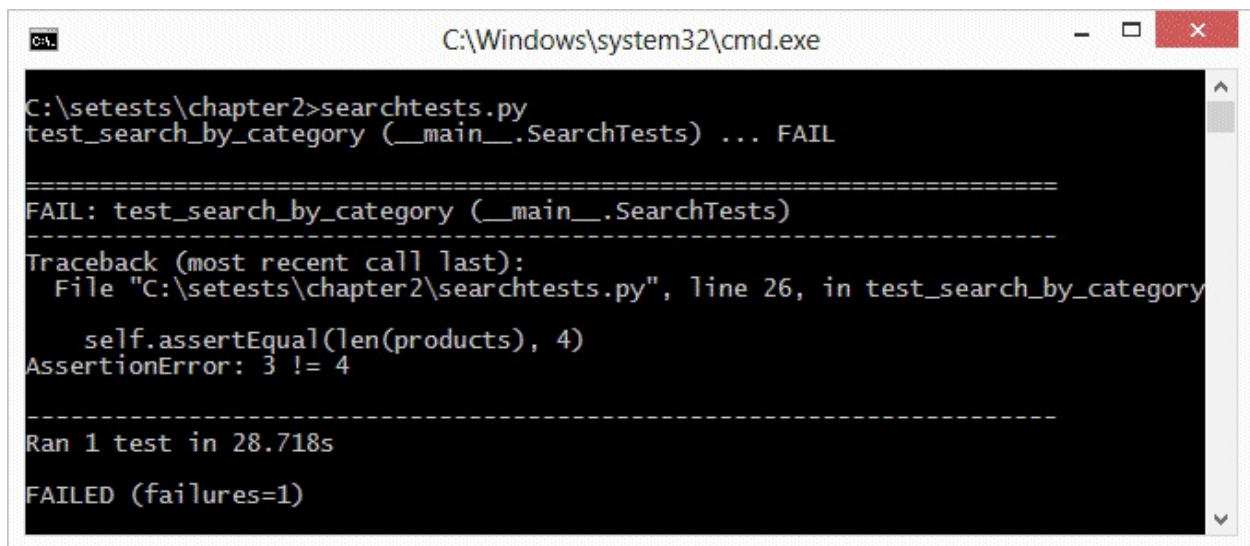
```
python searchtests.py
```

测试运行结束后，unittest会把测试结果和概要展示在控制台，如下图所示。



```
C:\Windows\system32\cmd.exe
C:\setests\chapter2>searchtests.py
test_search_by_category (__main__.SearchTests) ... ok
-----
Ran 1 test in 28.354s
OK
```

除了测试结果概要外，当一个测试用例执行失败，针对每个失败，测试结果概要都会通过生成文本信息来展示具体哪里有错误。通过下面的截图，可以看到当我们修改预期结果值后会发生些什么。

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the execution of a Python script "searchtests.py" in the directory "C:\setests\chapter2". The test "test_search_by_category" fails. The output shows a traceback indicating an "AssertionError: 3 != 4" at line 26 of "searchtests.py". The test took 28.718 seconds to run, and the overall result is "FAILED (failures=1)".

```
C:\Windows\system32\cmd.exe
C:\setests\chapter2>searchtests.py
test_search_by_category (__main__.SearchTests) ... FAIL

=====
FAIL: test_search_by_category (__main__.SearchTests)
=====
Traceback (most recent call last):
  File "C:\setests\chapter2\searchtests.py", line 26, in test_search_by_category
    self.assertEqual(len(products), 4)
AssertionError: 3 != 4

=====
Ran 1 test in 28.718s

FAILED (failures=1)
```

上图展示了具体是哪个测试方法执行失败，通过打印信息可以追踪具体导致失败的代码。另外，失败自身也会以`AssertionError`形式显示，例子中的预期结果和实际结果并不匹配。

2.1.1.5 添加其他测试

我们可以用一组测试来构建一个测试类，这样有助于为一个特定功能创建一组更合乎逻辑的测试。下面为测试类添加其他的测试。规则很简单，新的测试方法命名也要以`test`开头，如下列代码。

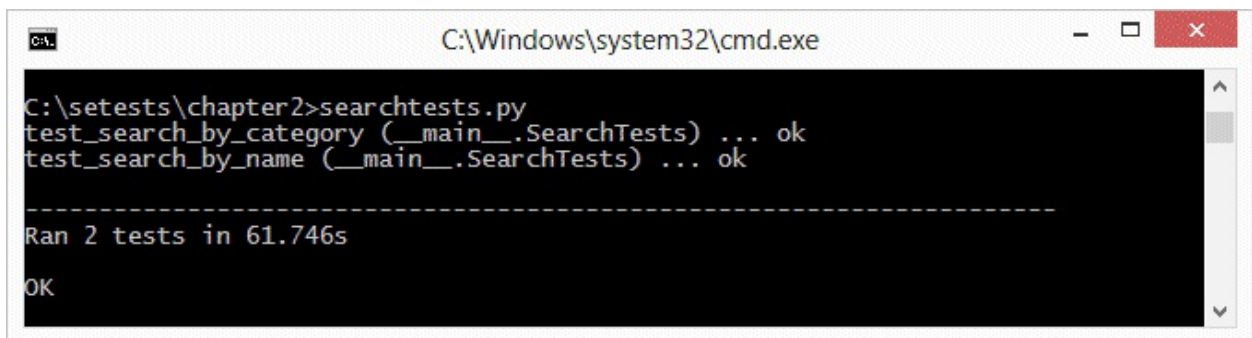
```
def test_search_by_name(self):
    # get the search textbox
```

```
self.search_field = self.driver.find_element_by_name("q")
self.search_field.clear()

# enter search keyword and submit
self.search_field.send_keys("salt shaker")
self.search_field.submit()

# get all the anchor elements which have
# product names displayed
# currently on result page using
# find_elements_by_xpath method
products = self.driver.find_elements_by_xpath(
    ("//h2[@class='product-name']/a")
self.assertEqual(1, len(products))
```

运行这个测试类将能看到两个Firefox的实例打开和关闭，这正是setUp()方法和tearDown()方法针对每个测试方法都要执行产生的结果，如下图所示。

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The text shows the execution of a Python script: "C:\setests\chapter2>searchtests.py". Below this, two test results are shown: "test_search_by_category (__main__.SearchTests) ... ok" and "test_search_by_name (__main__.SearchTests) ... ok". A dashed line separates these from the summary "Ran 2 tests in 61.746s". At the bottom, there is an "OK" message. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
C:\Windows\system32\cmd.exe

C:\setests\chapter2>searchtests.py
test_search_by_category (__main__.SearchTests) ... ok
test_search_by_name (__main__.SearchTests) ... ok

-----
Ran 2 tests in 61.746s

OK
```

2.1.2 类级别的setUp()方法和tearDown()

方法

在前面的例子中，我们通过setUp()方法为每个测试方法都创建了一个Firefox实例，并且在每个测试方法执行结束后都要关闭实例。能否让各个测试方法共用一个Firefox实例，而不要每次都创建一个新的实例呢？这可以通过使用setUpClass()方法和tearDownClass()方法及@classmethod标识来实现。这两个方法使我们可以类级别来初始化数据，替代了方法级别的初始化，这样各个测试方法就可以共享这些初始化数据。在下面的例子中，代码修改为调用setUpClass()方法和tearDownClass()方法并且加上@classmethod标识。

```
import unittest
from selenium import webdriver

class SearchTests(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # create a new Firefox session
        cls.driver = webdriver.Firefox()
        cls.driver.implicitly_wait(30)
```



```
cls.driver.maximize_window()

# navigate to the application home page
cls.driver.get("http://demo.magentocommerce.com
/")

cls.driver.title

def test_search_by_category(self):
    # get the search textbox
    self.search_field = self.driver.find_element_by
_name("q")
    self.search_field.clear()

    # enter search keyword and submit
    self.search_field.send_keys("phones")
    self.search_field.submit()

    # get all the anchor elements which have produc
t names
    # displayed currently on result page using
    # find_elements_by_xpath method
    products = self.driver.find_elements_by_xpath
        ("//h2[@class='product-name']/a")
    self.assertEqual(2, len(products))

def test_search_by_name(self):
    # get the search textbox
    self.search_field = self.driver.find_element_by
_name("q")
    self.search_field.clear()

    # enter search keyword and submit
    self.search_field.send_keys("salt shaker")
    self.search_field.submit()
```

```
# get all the anchor elements which have product names
# displayed currently on result page using
# find_elements_by_xpath method
products = self.driver.find_elements_by_xpath(
    ("//h2[@class='product-name']/a")
self.assertEqual(1, len(products))

@classmethod
def tearDownClass(cls):
    # close the browser window
    cls.driver.quit()

if __name__ == '__main__':
    unittest.main()
```

运行这个测试将看到仅创建一个Firefox实例，所有的测试都用同一个实例。



要了解更多关于@classmethod标识的信息，参考：<https://docs.python.org/2/library/functions.html#classmethod>。

2.1.3 断言

unittest的TestCase类提供了很多实用的方法来校验预期结果和程序返回的实际结果是否一致。这些方法要求必须满足某些条件才能继续执行接下来的测试。大致有3种这样的方法，各覆盖一个特定类型的条件，例如等价校验、逻辑校验和异常校验。如果给定的断言通过了，接下来的测试代码将会执行；相反，将会导致测试立即停止并且给出异常信息。

unittest提供了所有的标准xUnit 断言方法。下表列出了一些在本书后面将要用到的重要方法。

方 法	校 验 条 件	应 用 实 例
assertEqual(a, b [,msg])	a == b	这些方法校验a和b是否相等，msg对象是用来说明失败原因的消息。这对于验证元素的值和属性等是非常有用的。例如： assertEqual(element.text, "10")
assertNotEqual(a, b[,msg])	a != b	
assertTrue(x[,msg]))	bool(x) is True	这些方法校验给出的表达式是True还是False。例如，校验一个元素是否出现在页面，我们可以用下面的方法： assertTrue(element.is_displayed())
assertFalse(x[,msg]))	bool(x) is False	
assertIsNot(a, b[,msg]))	a is not b	
assertRaises(exc, fun, *args, **kwargs)	fun(*args, **kwargs) raises exc	这些方法校验特定的异常是否被具体的测试步骤抛出，用到该方法的一种可能情况是： NoSuchElementException
assertRaisesRegexp(exc, r, fun, *args, **kwargs)	fun(*args, **kwargs) raises exc and the message matches	

	regex r	
assertAlmostEqual(a, b)	round(a-b, 7) == 0	这些方法用于检查数值，在检查之前会按照给定的精度把数字四舍五入。这有助于统计由于四舍五入产生的错误和其他由于浮点运算产生的问题
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0	
assertGreater(a, b)	a > b	这些方法类似于assertEqual()方法，是为逻辑判定条件设计的
assertGreaterEqual(a, b)	a >= b	
assertLess(a, b)	a < b	
assertLessEqual(a, b)	a <= b	
assertRegexpMatches(s, r)	r.search(s)	这些方法检查文本是否符合正则匹配
assertNotRegexpMatches(s, r)	not r.search(s)	
assertMultiLineEqual(a, b)	strings	此方法是assertEqual()的一种特殊形式，为多行字符串设计。等值校验和其他单行字符串校验一样，但是默认失败信息经过优化以后可以展示具体值之间的差别
assertListEqual(a, b)	lists	此方法校验两个list是否相等，对于下拉列表选项字段的校验是非常有用的
fail()		此方法是无条件的失败。在别的assert方法不好用的时候，也可用此方法来创建定制的条件块

2.1.4 测试套件

应用unittest的TestSuites特性，可以将不同的测试组成一个逻辑组，然后设置统一的测试套件，并通过一个命令来执行测试。这都是通过TestSuites、TestLoader和TestRunner类来实现的。

在了解TestSuites的细节之前，我们为例子程序添加一个新的测试，用于校验主页。我们将把新加

的测试和之前的测试放到一个测试组件中，详见下面代码。

```
import unittest
from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException
from selenium.webdriver.common.by import By
from __builtin__ import classmethod

class HomePageTest(unittest.TestCase):
    @classmethod
    def setUp(cls):
        # create a new Firefox session """
        cls.driver = webdriver.Firefox()
        cls.driver.implicitly_wait(30)
        cls.driver.maximize_window()
        # navigate to the application home page """
        cls.driver.get("http://demo.magentocommerce.com/")

    def test_search_field(self):
        # check search field exists on Home page
        self.assertTrue(self.is_element_present(By.NAME, "q"))

    def test_language_option(self):
        # check language options dropdown on Home page
        self.assertTrue(self.is_element_present(
            By.ID, "select-language"))

    def test_shopping_cart_empty_message(self):
        # check content of My Shopping Cart block on Ho
```

me page

```
        shopping_cart_icon = \
            self.driver.find_element_by_css_selector
                ("div.header-minicart span.icon")
        shopping_cart_icon.click()

        shopping_cart_status = \
            self.driver.find_element_by_css_selector
                ("p.empty").text
        self.assertEqual("You have no items in your shopping cart.", shopping_cart_status)

        close_button = self.driver.find_element_by_css_selector
            ("div.minicart-wrapper a.close")
        close_button.click()

    @classmethod
    def tearDown(cls):
        # close the browser window
        cls.driver.quit()

    def is_element_present(self, how, what):
        """
        Utility method to check presence of an element
on page
        :params how: By locator type
        :params what: locator value
        """
        try: self.driver.find_element(by=how, value=what)

        except NoSuchElementException, e: return False
        return True

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

我们将用TestSuite类来定义和执行测试套件。我们可以把多个测试加入到一个测试套件中去。除了TestSuite类，我们还可以用TestLoader和TextTestRunner来创建和运行测试套件，举例如下。

```
import unittest
from searchtests import SearchTests
from homepagetests import HomePageTest

# get all tests from SearchProductTest and HomePageTest
class
search_tests = unittest.TestLoader().loadTestsFromTestCase
(SearchTests)
home_page_tests = unittest.TestLoader().loadTestsFromTestCase
(HomePageTest)

# create a test suite combining search_test and home_page_test
smoke_tests = unittest.TestSuite([home_page_tests, search_tests])

# run the suite
unittest.TextTestRunner(verbosity=2).run(smoke_tests)
```

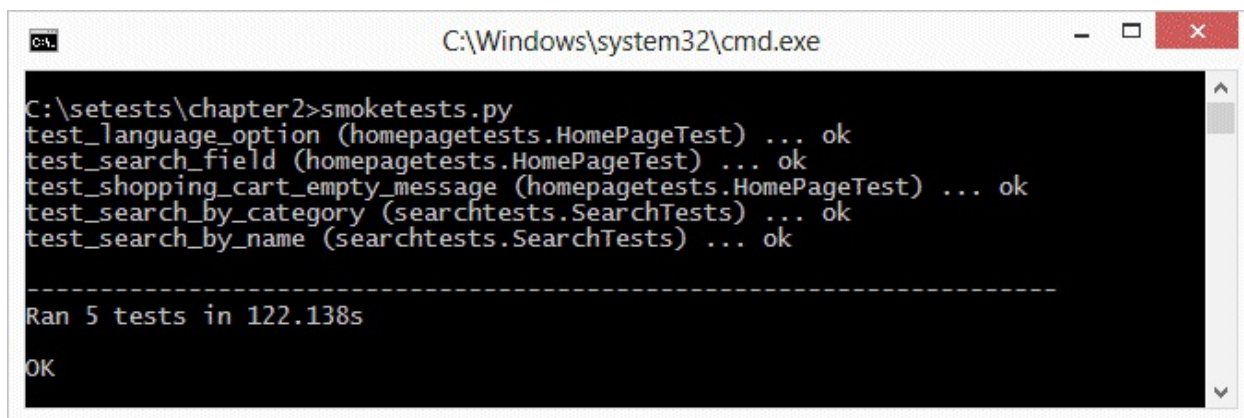
使用TestLoader类，我们将得到指定测试文件中的所有测试方法且用于创建测试套件。

TestRunner类将通过调用测试套件来执行文件中所有的测试。

我们可以通过下面的命令运行新的测试套件文件。

```
python smoketests.py
```

这将运行SearchProductTest类和HomePageTest类中的所有测试并且通过命令行形式生成下图这样的测试输出。



```
C:\Windows\system32\cmd.exe

C:\setests\chapter2>smoketests.py
test_language_option (homepagetests.HomePageTest) ... ok
test_search_field (homepagetests.HomePageTest) ... ok
test_shopping_cart_empty_message (homepagetests.HomePageTest) ... ok
test_search_by_category (searchtests.SearchTests) ... ok
test_search_by_name (searchtests.SearchTests) ... ok

-----
Ran 5 tests in 122.138s

OK
```


2.2 生成HTML格式的测试报告

`unittest`在命令行输出测试结果。你可能需要生成一个所有测试的执行结果作为报告或者把测试结果发给相关人员。给相关人员发送命令行日志不是一个明智的选择。他们需要格式更加友好的测试报告，既能够查看测试结果的概况，也能够深入查看报告细节。`unittest`没有相应的内置模块可以生成格式友好的报告，我们可以应用Wai Yip Tung编写的`unittest`的扩展`HTMLTestRunner`来实现。从下面的网址可以获取更多关于`HTMLTestRunner`的信息并可以下载说明文档：

<https://pypi.python.org/pypi/HTMLTestRunner>。



`HTMLTestRunner`扩展可以在本书的附件源代码中找到。

我们将在测试中使用HTMLTestRunner来生成漂亮的测试报告。通过修改在本章前面涉及的测试套件文件来添加HTMLTestRunner支持。我们需要创建一个包含实际测试报告的输出文件，需要配置HTMLTestRunner选项和运行测试，具体如下。

```
import unittest
import HTMLTestRunner
import os
from searchtests import SearchTests
from homepagetests import HomePageTest

# get the directory path to output report file
dir = os.getcwd()

# get all tests from SearchProductTest and HomePageTest
class
search_tests = unittest.TestLoader().loadTestsFromTestCase(SearchTests)
home_page_tests = unittest.TestLoader().loadTestsFromTestCase(HomePageTest)

# create a test suite combining search_test and home_page_test
smoke_tests = unittest.TestSuite([home_page_tests, search_tests])

# open the report file
```

```
outfile = open(dir + "\SmokeTestReport.html", "w")

# configure HTMLTestRunner options
runner = HTMLTestRunner.HTMLTestRunner(
    stream=outfile,
    title='Test Report',
    description='Smoke Tests'
)

# run the suite using HTMLTestRunner
runner.run(smoke_tests)
```

执行该测试套件，HTMLTestRunner像unittest的默认测试执行器一样运行所有的测试。在用例执行的最后，它将生成测试报告文件，如下图所示。

Test Report

←

→

↻

file:///C:/setests/chapter2/SmokeTestReport.html

☆

»

≡

Apps

Selenium

Agile

Symantec

Testing

Apple/iOS

Web

» Other bookmarks

Test Report

Start Time: 2014-09-08 11:26:39
Duration: 0:01:51.986000
Status: Pass 5

Smoke Tests

Show [Summary](#) [Failed](#) [All](#)

Test Group/Test case	Count	Pass	Fail	Error	View
homepagetests.HomePageTest	3	3	0	0	Detail
test_language_option				pass	
test_search_field				pass	
test_shopping_cart_empty_message				pass	
searchtests.SearchTests	2	2	0	0	Detail
test_search_by_category				pass	
test_search_by_name				pass	
Total	5	5	0	0	

2.3 章节回顾

本章我们学习了如何使用unittest来编写和运行基于Selenium WebDriver的测试脚本。我们使用包含setUp()方法和tearDown()方法的TestCase类来创建测试。我们还可以添加断言来验证预期结果和实际结果是否一致。

我们也学习了如何使用unittest支持的不同类型的断言。我们实现了测试套件，它提供了把不同的测试用例组成逻辑分组的能力。最后，我们使用HTMLTestRunner来生成格式友好的HTML格式的测试报告。

在下一章中，我们将学习如何定义和使用定位器来与页面中的不同类型的HTML元素进行交互。

第3章 元素定位

Web应用以及包含超文本标记语言（**HTML**）、层叠样式表（**CSS**）、JavaScript脚本的Web页面；基于用户的操作行为诸如跳转到指定的统一资源定位（**URL**）网站，或是单击提交按钮，浏览器向Web服务器发送请求；Web服务器响应请求，返回给浏览器HTML以及相关的JavaScript、CSS、图片等资源；浏览器使用这些资源生成Web页面，其中包含Web各种视觉元素，例如文本框、按钮、标签页、图表、复选框、单选按钮、列表、图片等。上面列举的这些，我们普通用户并不用关心，放心交给HTML去组织并且最终呈现在浏览器里即可。这些视觉元素或控件都被Selenium称为页面元素（**WebElements**）。

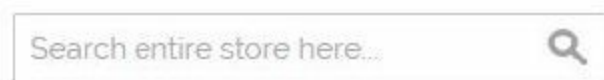
本章包含以下主题：

- 理解更多Selenium WebDriver定位元素的方法；
- 理解利用浏览器开发者模式辅助定位元素的方法；
- 定位元素的方法包括通过ID定位、name定位、class属性定位以及利用XPath和CSS选择器定位；
- 通过各种find_element_by的方法查找元素，以便使用Selenium WebDriver与之自动化交互。

当我们想让Selenium自动地操作我们的浏览器，就必须告诉Selenium如何去定位某个元素或一组元素，可以通过编程的方式去模拟用户操作。每个元素有着不同的标签名和属性值，Selenium提供了多种选择与定位元素的方法。

我们如何获取这些信息呢？大家都知道，Web页面是由HTML、CSS和JavaScript等组成的。我们

可以通过查看页面源文件的方式了解这些文本信息，进而可以找到我们想要的tag标签，了解与之对应的元素是如何交互的、如何定义属性与属性值的，以及页面的结构。下面展示了一个我们正在测试的场景，这是一个常见的搜索功能，包括搜索框与搜索按钮（放大镜图标）。



让我们看一下其对应的HTML脚本。

```
<form id="search_mini_form" action=
  "http://demo.magentocommerce.com/catalogsearch/result
/"
  method="get">
  <div class="form-search">
    <label for="search">Search:</label>
    <input id="search" type="text" name="q" value="
"
      class="input-text" maxlength="128" />
    <button type="submit" title="Search"
      class="button"><span><span>Search</span></spa
n></button>
    <div id="search_autocomplete" class="search-
autocomplete"></div>
    <script type="text/javascript">
```



```
//
    var searchForm = new Varien.searchForm
        ('search_mini_form', 'search', 'Search en
tire store
        here...');
    searchForm.initAutocomplete
        ('http://demo.magentocommerce.com
        /catalogsearch/ajax/suggest/',
        'search_autocomplete');
//]]&gt;
&lt;/script&gt;
&lt;/div&gt;
&lt;/form&gt;</pre></div><div data-bbox="111 424 855 592" data-label="Text"><p>我们发现类似搜索框、搜索按钮这样的元素，都是采用内嵌在&lt;form&gt;标签内的&lt;input&gt;标签来实现，标记则用了&lt;label&gt;标签来实现。另外，JavaScript代码写在了&lt;script&gt;标签内。</p></div><div data-bbox="111 633 776 710" data-label="Text"><p>其中搜索框&lt;input&gt;标签中包含id、type、name、value、class和maxlength属性的定义。</p></div><div data-bbox="121 751 773 796" data-label="Text"><pre>&lt;input id="search" type="text" name="q" value=""
class="input-text" maxlength="128" /&gt;</pre></div><div data-bbox="183 835 879 864" data-label="Text"><p>我们可以在浏览器窗口右键单击，在快捷菜单</p></div>
```

中选择查看源文件选项，在弹出的窗口中可以显示HTML文件与JavaScript脚本。



如果你对查看HTML、CSS和JavaScript感到生疏的话，可以查看相关网站，可以帮助你更快地识别WebDriver所需的元素的位置。

3.1 借助浏览器开发者模式定位

在使用Selenium测试之前，我们通常会先去查看页面源代码，借助工具可以帮助我们了解页面结构。值得庆幸的是，目前绝大多数的浏览器都内置有相关插件，能够快速、简洁地展示各类元素的属性定义、DOM结构、JavaScript代码块、CSS样式等属性。接下来我们一起学习这类工具的细节以及使用方法。

3.1.1 用火狐浏览器Firebug插件检查页面元素

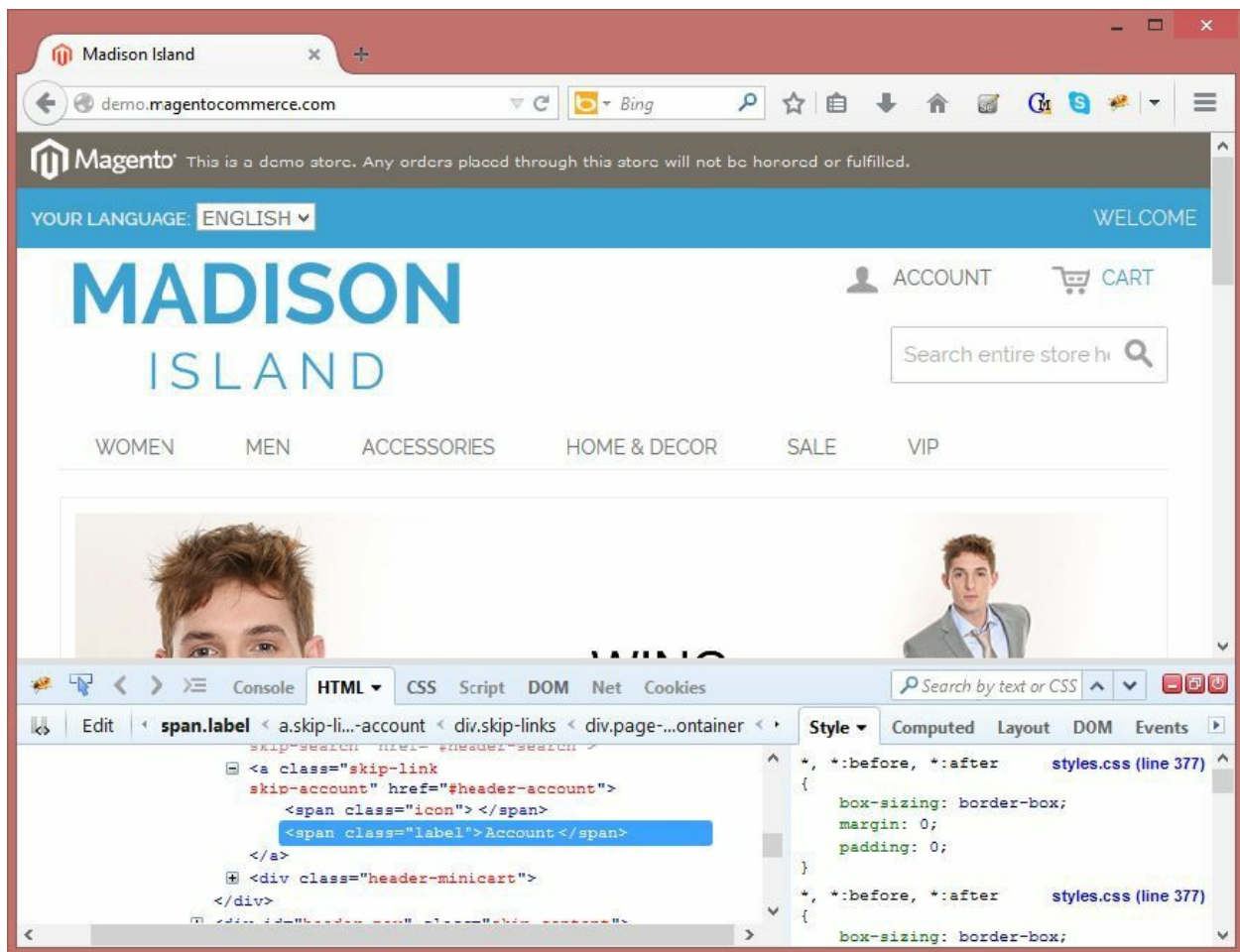
较新版本的火狐浏览器尽管自带有页面分析工具，然而，我们还是建议大家使用功能更强大的Firebug插件。

(1) 你可以通过下列地址，下载并安装Firebug插件。

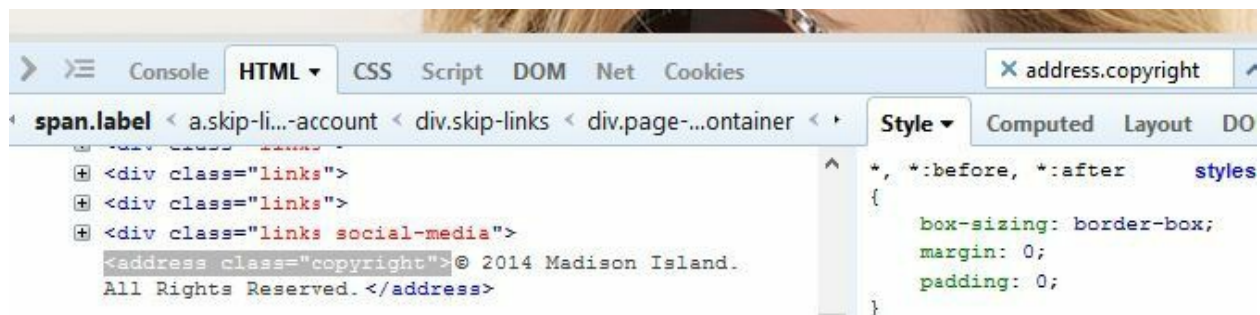
<https://addons.mozilla.org/en-us/firefox/addon/firebug/>

(2) 尝试用Firebug，在页面上移动鼠标至希望获取的元素，然后右键单击，弹出快捷菜单。

(3) 选择使用**Firebug**查看元素。此时火狐浏览器下方会显示HTML代码树窗口并定位到所选的元素上，如下图所示。



(4) 我们还可以使用Firebug的XPath或CSS选择器，通过Firebug弹出窗口自带的检索功能，只要键入想要查找的关键字，就可以高亮显示与之匹配的元素，如下图所示。

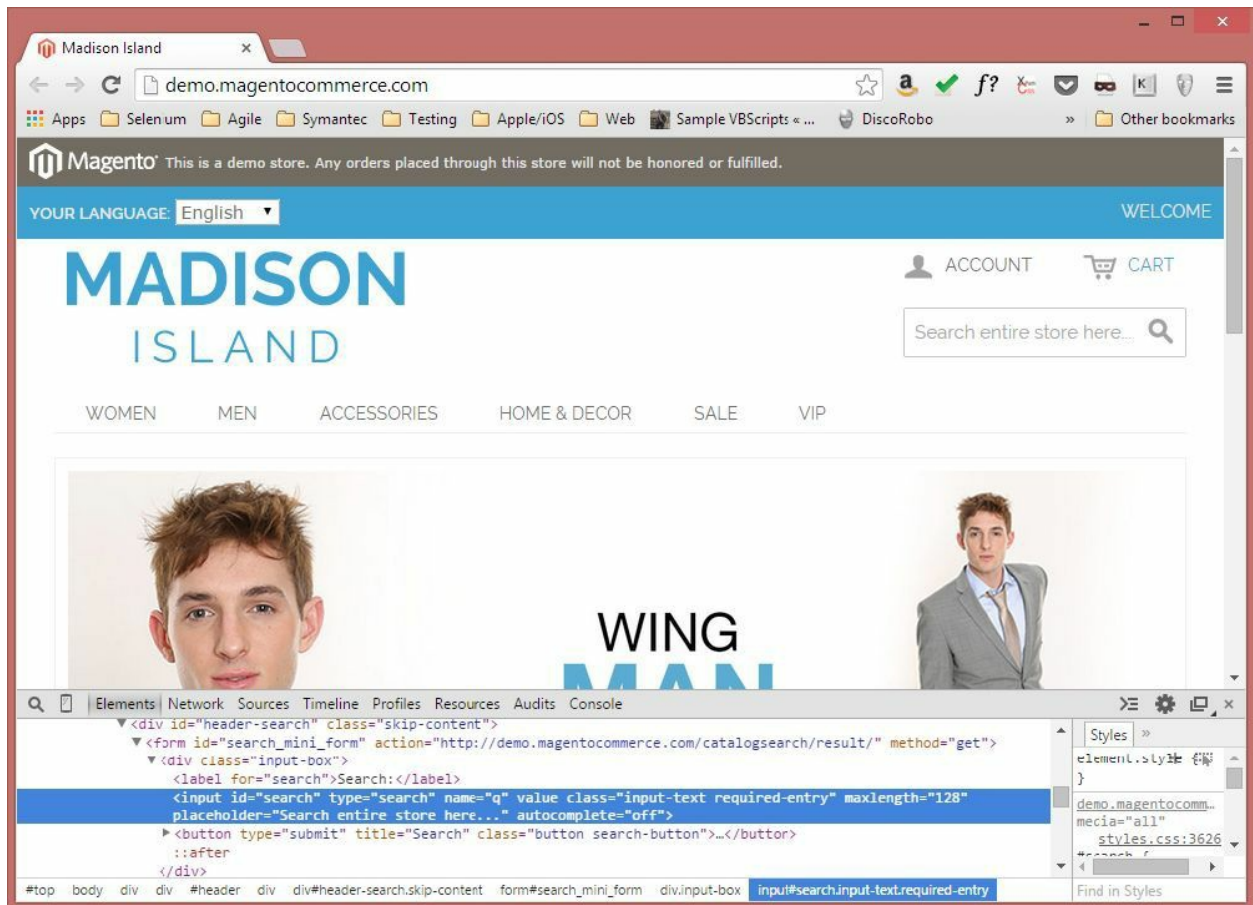


3.1.2 用谷歌Chrome浏览器检查页面元素

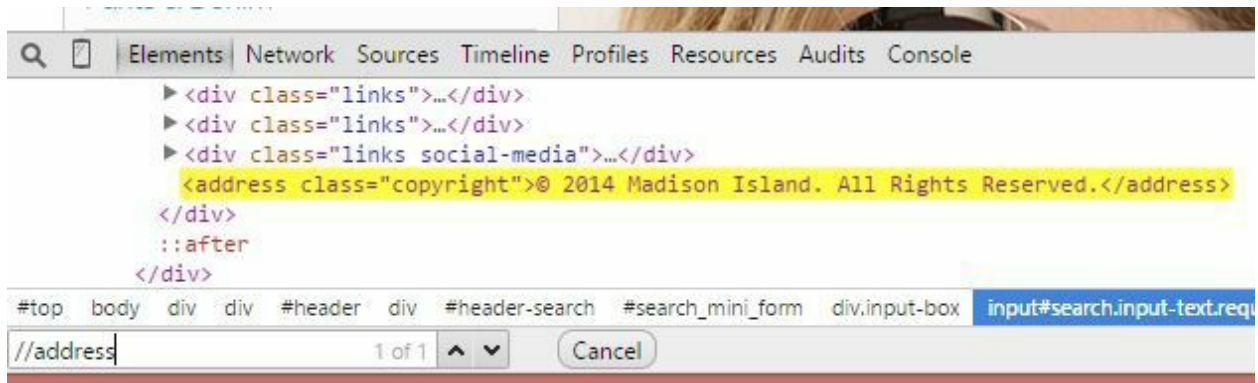
谷歌Chrome浏览器也自带有页面分析的功能。你可以通过以下步骤来检查页面元素。

(1) 首先移动鼠标光标到期望的元素上，然后右键单击，在弹出的快捷菜单中，选择检查(N)选项。

在浏览器的下方，将显示类似Firebug的开发者工具窗口，如下图所示。



(2) 同样类似Firebug，当我们需要使用XPath或CSS选择器时，在开发者工具下的Elements窗口中，按Ctrl+F键，将会显示搜索框。一旦输入XPath或CSS表达式，Firebug就会高亮显示与之匹配的元素，如下图所示。

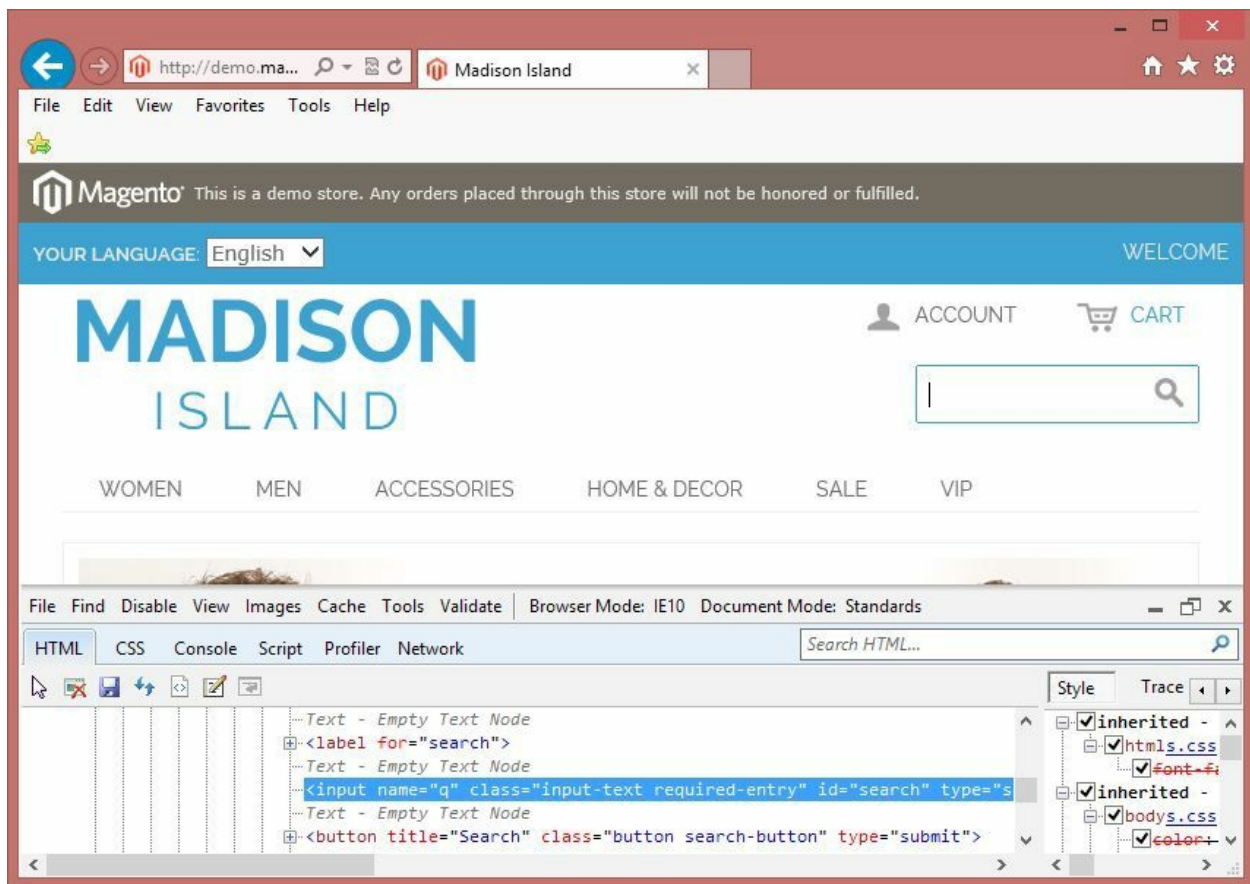


3.1.3 用IE浏览器检查页面元素

微软公司的IE浏览器也自带有页面分析的功能。你可以通过以下步骤来检查页面元素。

(1) 按F12键，在浏览器下方显示开发者工具窗口。

(2) 在开发者工具窗口选择“箭头”按钮，然后单击页面中期望获取的元素，在开发者工具窗口将高亮显示对应的HTML代码树，如下图所示。



通过上述工具，可以非常有效地帮助我们编写测试代码，以及执行与调试JavaScript脚本。

3.2 元素定位

我们必须告诉Selenium怎样去定位元素，用来模拟用户动作，或者查看元素的属性和状态，以便我们可以执行检查。例如，我们要搜索一个产品，首先要找到搜索框与搜索按钮，接着通过键盘输入要查询的关键字，最后用鼠标单击搜索按钮，提交搜索请求。

正如上述人工的操作步骤一样，我们也希望Selenium能模拟我们的动作，然而，Selenium并不能理解类似在搜索框中输入关键字或单击搜索按钮这样图形化的操作。所以需要我们程序化地告诉Selenium如何定位搜索框与搜索按钮，从而模拟键盘与鼠标的动作。

Selenium提供多种find_element_by 方法用于定位页面元素。这些方法根据一定的标准去查找元

素，如果元素被正常定位，那么WebElement实例将返回。反之，将抛出NoSuchElementException的异常。同时，Selenium还提供多种find_elements_by 方法去定位多个元素，这类方法根据所匹配的值，搜索并返回一个list数组（元素）。

Selenium提供8种find_element_by 方法用于定位元素。接下来的部分，我们将逐一介绍方法细节，如下表所示。

方 法	描 述	参 数	示 例
find_element_by_id(id)	通过元素的ID属性值来定位元素	id: 元素的ID	driver.find_element_by_id('search')
find_element_by_name(name)	通过元素的name属性值来定位元素	name: 元素的name	driver.find_element_by_name('q')

find_element_by_class_name(name)	通过元素的class名来定位元素	name: 元素的类名	driver.find_element_by_class_name('i
find_element_by_tag_name(name)	通过元素的tag name来定位元素	name: tag name	driver.find_element_by_tag_name('in
find_element_by_xpath(xpath)	通过XPath来定位元素	XPath: 元素的XPath	driver.find_element_by_xpath('///form
find_element_by_css_selector(css_selector)	通过CSS选择器来定位元素	css_selector: 元素的CSS选择器	driver.find_element_by_css_selector('
find_element_by_link_text(link_text)	通过元素标签对之间的文本信息来定位元素	link_text: 文本信息	driver.find_element_by_link_text('Lo
	通过元素标签		

<code>find_element_by_partial_link_text(link_text)</code>	对之间的部分文本信息来定位元素	<code>link_text</code> : 部分文本信息	<code>driver.find_element_by_partial_link_t</code>
---	-----------------	---------------------------------	--

`find_elements_by` 方法能按照一定的标准返回一组元素，具体见下表。

方 法	描 述	参 数	示 例
<code>find_elements_by_id(id_)</code>	通过元素的ID属性值来定位一组元素	<code>id</code> : 元素的ID	<code>driver.find_elements_by_id('product'</code>
<code>find_elements_by_name(name)</code>	通过元素的name属性值来定位一组元素	<code>name</code> : 元素的name	<code>driver.find_elements_by_name('prod</code>
<code>find_elements_by_class_name(name)</code>	通过元素的class名来定位	<code>name</code> : 元素的类名	<code>driver.find_elements_by_class_name</code>

	一组元素		
find_elements_by_tag_name(name)	通过元素的tag name来定位一组元素	name: tag name	driver.find_elements_by_tag_name('i
find_elements_by_xpath(xpath)	通过XPath来定位一组元素	XPath: 元素的XPath	driver.find_elements_by_xpath("//div
find_elements_by_css_selector(css_selector)	通过CSS选择器来定位一组元素	css_selector: 元素的CSS选择器	driver.find_elements_by_css_selecto
find_elements_by_link_text(text)	通过元素标签对之间的文本信息来定位一组元素	text: 文本信息	driver.find_elements_by_link_text('L
	通过元素标签对之间		

<code>find_elements_by_partial_link_text(link_text)</code>	的部分 文本信 息来定 位一组 元素	<code>link_text</code> : 部 分文本信息	<code>driver.find_elements_by_partial_link</code>
--	--------------------------------	-------------------------------------	---

3.2.1 ID定位

通过ID查找元素是查找页面上元素的最佳方法。`find_element_by_id()`和`find_elements_by_id()`方法返回与ID属性值匹配的一个元素或一组元素。

`find_element_by_id()`方法返回的是与ID属性值匹配的第一元素，如果没有元素与之匹配，则抛出`NoSuchElementException`异常。

如下图所示，我们尝试来定位搜索框。



通过查看HTML，我们可以看到搜索框的ID值被定义为search。

```
<input id="search" type="text" name="q" value=""  
class="input-text" maxlength="128" autocomplete="off">
```

接下来我们使用`find_element_by_id()`方法，`id`值为`search`来定位搜索框，同时检查`maxlength`的属性值。

```
def test_search_text_field_max_length(self):  
    # get the search textbox  
    search_field = self.driver.find_element_by_id("search")  
  
    # check maxlength attribute is set to 128  
    self.assertEqual("128", search_field.get_attribute("maxlength"))
```

此外，如果使用`find_elements_by_id()`方法，那么将返回匹配ID值的所有元素。

3.2.2 name定位

通过`name`定位是另外一个常用的查找元素的方式。`find_element_by_name()`和`find_elements_by_name()`方法可以通过匹配`name`值

来定位单个或一组元素。同样，`name`值匹配成功可返回定位的元素；反之，则抛出 `NoSuchElementException` 的异常。

回到之前的例子，我们可以用匹配`name`属性值的方式来替换ID值的匹配，同样可以定位到搜索框。

```
# get the search textbox
self.search_field = self.driver.find_element_by_name("q")
```

此外，如果使用`find_elements_by_name()`方法，那么将返回匹配`name`值的所有元素。

3.2.3 class定位

除了使用ID和`name`属性，我们还可通过`class`属性来定位元素。`class`用来关联CSS中定义的属性。`find_element_by_class_name()`和 `find_elements_by_class_name()`方法可以通过匹配

class属性来定位单个或一组元素。同样，class值匹配成功可返回定位的元素；反之，则抛出NoSuchElementException的异常。



通过对元素ID、name和class属性来查找元素是最为普遍和快捷的方法。此外，Selenium WebDriver还提供了其他一些方法用于定位元素，在接下来的段落中会有详细介绍。

同样的场景（见下图），我们可以尝试用find_element_by_class_name() 的方法来定位元素。



搜索按钮（放大镜图标）在HTML中是用<button>标签（元素）以及对应的class属性与属性

值定义的，具体如下。

```
<button type="submit" title="Search"  
    class="button"><span><span>Search</span></span></butt  
on>
```

下面我们用class属性值来定位搜索按钮，代码如下。

```
def test_search_button_enabled(self):  
    # get Search button  
    search_button = self.driver.find_element_by_class_name  
    ("button")  
  
    # check Search button is enabled  
    self.assertTrue(search_button.is_enabled())
```

此外，如果使用find_elements_by_class_name()方法，那么将返回匹配class属性值的所有元素。

3.2.4 tag定位

find_element_by_tag_name()和find_elements_by_tag_name()方法是通过HTML

页面中tag name匹配的方式来定位元素的。这些方法跟JavaScript中的DOM方法getElementsByTagName()类似。同样，通过成功匹配tag name可以返回定位的元素；反之，则抛出NoSuchElementException的异常。

这个方法在某些特定的场景下格外有用，例如，我们可以通过<tr>的tag name一次定位页面的table中所有的行数据（元素）。

如下图所示的几张banner图，通过查看HTML代码，我们可以看出包含有或者的标签。



这几张banner图采用无序列表标签内嵌图像标签来实现。

```
<ul class="promos">
```

```
<li>
  <a href="http://demo.magentocommerce.com/home-decor.html">
    
  </a>
</li>
<li>
  <a href="http://demo.magentocommerce.com/vip.html">
    
  </a>
</li>
<li>
  <a href="http://demo.magentocommerce.com/accessories/bags-luggage.html">
    
  </a>
</li>
</ul>
```

我们使用find_elements_by_tag_name()方法来定位所有的banner图片。首先我们用find_element_by_class_name()方法定位这一组banner

图，然后用find_elements_by_tag_name()方法去匹配的tag name，最后把结果返回给banners对象。

```
def test_count_of_promo_banners_images(self):
    # get promo banner list
    banner_list = self.driver.find_element_by_class_name("promos")

    # get images from the banner_list
    banners = banner_list.find_elements_by_tag_name("img")

    # check there are 20 tags displayed on the page
    self.assertEqual(2, len(banners))
```

3.2.5 XPath定位

XPath是一种在XML文档中搜索和定位元素的查询语言。几乎所有的浏览器都支持XPath。同样，Selenium也可以通过XPath的方式在Web页面上定位元素。

当我们发现通过ID、name或class属性值都无法

定位元素时，不妨尝试用XPath的方式。我们可以灵活地运用绝对或相对路径定位，也可以通过除ID、name以外的其他属性来定位，甚至还可以通过属性值的一部分（如starts-with()、contains()和ends-with()）来帮助我们定位。



了解更多关于XPath的知识，可访问相关网站了解。

想要了解更多关于XPath定位的知识，可以参考《Selenium Testing Tools Cookbook》，Packt Publishing。

接下来，我们可以使用find_element_by_xpath()和find_elements_by_xpath()方法来定位元素了。例如，我们通过之前的广告banner图，单击banner图

片进入对应的页面。



上图是名为“Shop Private Sales”的banner图，在的tag下，其中代码并不包含ID、name或class属性等信息，且这个页面还包含很多其他的，所以我们不能通过传统的方法如findbytag_name()简单地定位了。

```
<ul class="promos">
  ...
  <li>
    <a href="http://demo.magentocommerce.com/vip.html"
">
      
    </a>
  </li>
  ...
</ul>
```

我们尝试使用find_element_by_xpath()方法，用

标签下的 alt 属性值来定位我们要找的元素。

代码如下。

```
def test_vip_promo(self):
    # get vip promo image
    vip_promo = self.driver.\
        find_element_by_xpath("//img[@alt='Shop Private Sales - Members Only']")

    # check vip promo logo is displayed on home page
    self.assertTrue(vip_promo.is_displayed())
    # click on vip promo images to open the page
    vip_promo.click()
    # check page title
    self.assertEqual("VIP", self.driver.title)
```

此外，如果使用find_elements_by_xpath() 方法，那么将返回匹配XPath查询到的所有元素。

3.2.6 CSS选择器定位

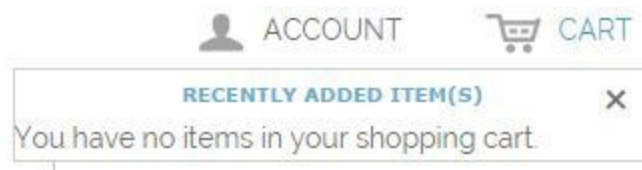
CSS（层叠样式表）是一种用于页面设计（HTML）与表现的文件样式，是一种计算机语言，能灵活地为页面提供各种样式风格。CSS使用

选择器为页面元素绑定属性（如ID、class、type、attribute、value等）。

类似XPath，Selenium也可以利用CSS选择器的特性，用于帮助我们来定位元素。如果想进一步了解关于CSS选择器的一些知识，请访问相关网站了解。

下面介绍find_element_by_css_selector()和find_elements_by_css_selector()两种方法。

回到首页的例子，可以看到购物车按钮，单击这个按钮，将进入购物车页面。如果此时没有添加任何商品，那么系统会提示“你还没有添加商品到购物车”，如下图所示。



HTML代码如下。

```
<div class="minicart-wrapper">
<p class="block-subtitle">
    Recently added item(s)
    <a class="close skip-link-close" href="#" title="Close">
        x</a>
</p>
    <p class="empty">You have no items in your shopping
    cart.
    </p>
</div>
```

我们设计测试程序来校验这个提示信息。首先我们将使用CSS选择器来定位购物车按钮，然后单击它，紧接着定位即将弹出的信息。

```
def test_shopping_cart_status(self):
    # check content of My Shopping Cart block on Home page
    # get the Shopping cart icon and click to open the
    # Shopping Cart section
    shopping_cart_icon = self.driver.\
        find_element_by_css_selector("div.header-minicart span.icon")
    shopping_cart_icon.click()

    # get the shopping cart status
    shopping_cart_status = self.driver.\
```

```
        find_element_by_css_selector("p.empty").text
        self.assertEqual("You have no items in your shopping
    cart.", shopping_cart_status)
    # close the shopping cart section
    close_button = self.driver.\
        find_element_by_css_selector("div.minicart-wrapper a.close")
    close_button.click()
```

从上面的测试脚本可以看出，我们使用了元素tag和class name来缩小获取购物车按钮的范围。

```
shopping_cart_icon = self.driver.\
    find_element_by_css_selector("div.header-minicart span.icon")
```

首先定位到tag 名为<div>的元素，然后接着“**.header-minicart**”的类名，其下面的标签下又有“.icon”的类名。

想要了解更多关于CSS选择器的知识，请参考《Selenium Testing Tools Cookbook》，Packt Publishing。

3.2.7 Link定位

`find_element_by_link_text()` 和 `find_elements_by_link_text()` 方法是通过文本链接来定位元素。如下示例。

(1) 定位首页上的Account链接，如下图所示，我们可以使用`find_element_by_link_text()`方法。



(2) 查看对应的HTML代码，具体如下。

```
<a href="#header-account" class="skip-link skip-account">
  <span class="icon"></span>
  <span class="label">Account</span>
</a>
```

(3) 编写测试脚本，先通过文本定位Account链接，然后单击查看是否能显示。

```
def test_my_account_link_is_displayed(self):
    # get the Account link
    account_link =
```

```
self.driver.find_element_by_link_text("ACCOUNT")

# check My Account link is displayed/visible in
# the Home page footer
self.assertTrue(account_link.is_displayed())
```

此外，如果使用find_elements_by_link_text()方法，那么将返回匹配文本的所有元素。

3.2.8 Partial link定位

find_element_by_partial_link_text()和find_elements_by_partial_link_text()两个方法是通过文本链接的一部分文本来定位元素的方法。如下示例。

(1) 同样是首页，有两个链接可以查看个人账户（Account）页面，一个是页面标头（header）部分的Account文字链接，另外一个页脚（footer）部分的My Account文字链接。

(2) 我们使用

`find_elements_by_partial_link_text()`方法，通过部分文本信息“Account”来定位，验证页面中的两个文本链接是否都能定位到（断言）。代码如下。

```
def test_account_links(self):
    # get the all the links with Account text in it
    account_links = self.driver.\
        find_elements_by_partial_link_text("ACCOUNT")

    # check Account and My Account link is displayed/vi
    sible in the Home page footer
    self.assertTrue(2, len(account_links))
```

3.3 方法实践

通过前面的介绍，我们尝试了很多种关于 `find_element_by` 的方法。接下来我们把这种类型的定位方法集成到同一个测试脚本中来。

(1) 创建一个名为 `homepagetest.py` 的 Python 脚本，整合之前我们创建的那些测试代码。

```
import unittest
from selenium import webdriver

class HomePageTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # create a new Firefox session
        cls.driver = webdriver.Firefox()
        cls.driver.implicitly_wait(30)
        cls.driver.maximize_window()

        #navigate to the application home page
        cls.driver.get('http://demo.magentocommerce.com/
')

    def test_search_text_field_max_length(self):
        # get the search textbox
        search_field = self.driver.
```



```

        find_element_by_id("search")

        # check maxlength attribute is set to 128
        self.assertEqual("128", search_field.get_attribute
te        ("maxlength"))

    def test_search_button_enabled(self):
        # get Search button
        search_button = self.driver.
            find_element_by_class_name("button")

        # check Search button is enabled
        self.assertTrue(search_button.is_enabled())

    def test_my_account_link_is_displayed(self):
        # get the Account link
        account_link =
            self.driver.find_element_by_link_text("ACCOUNT
")

        # check My Account link is displayed/visible in
        # the Home page footer
        self.assertTrue(account_link.is_displayed())

    def test_account_links(self):
        # get the all the links with Account text in it
        account_links = self.driver.\
            find_elements_by_partial_link_text("ACCOUNT")

        # check Account and My Account link is
        # displayed/visible in the Home page footer
        self.assertTrue(2, len(account_links))

    def test_count_of_promo_banners_images(self):
        # get promo banner list

```

```
banner_list = self.driver.\n    find_element_by_class_name("promos")\n\n# get images from the banner_list\nbanners = banner_list.\n    find_elements_by_tag_name("img")\n\n# check there are 3 banners displayed on the pag  
e\nself.assertEqual(2, len(banners))\n\ndef test_vip_promo(self):\n    # get vip promo image\n    vip_promo = self.driver.\n        find_element_by_xpath("//img[@alt=\n            'Shop Private Sales - Members Only']")\n\n    # check vip promo logo is displayed on home page\n\n    self.assertTrue(vip_promo.is_displayed())\n    # click on vip promo images to open the page\n    vip_promo.click()\n    # check page title\n    self.assertEqual("VIP", self.driver.title)\n\ndef test_shopping_cart_status(self):\n    # check content of My Shopping Cart block\n    # on Home page\n    # get the Shopping cart icon and click to\n    # open the Shopping Cart section\n    shopping_cart_icon = self.driver.\n        find_element_by_css_selector("div.header-\n            minicart span.icon")\n    shopping_cart_icon.click()
```

```
ping
    # get the shopping cart status
    shopping_cart_status = self.driver.\
        find_element_by_css_selector("p.empty").text
    self.assertEqual("You have no items in your shop
    cart.", shopping_cart_status)
    # close the shopping cart section
    close_button = self.driver.\
        find_element_by_css_selector("div.minicart-
        wrapper a.close")
    close_button.click()

    @classmethod
    def tearDownClass(cls):
        # close the browser window
        cls.driver.quit()

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

(2) 保存py文件，可以在命令行中直接执行。

```
python homepage_test.py
```

(3) 执行测试过程中，unittest会打印7组测试的执行结果（OK），如下图所示。

```
C:\Windows\system32\cmd.exe

c:\setests\chapter3>homepagetests.py
test_account_links (__main__.HomePageTest) ... ok
test_count_of_promo_banners_images (__main__.HomePageTest) ... ok
test_my_account_link_is_displayed (__main__.HomePageTest) ... ok
test_search_button_enabled (__main__.HomePageTest) ... ok
test_search_text_field_max_length (__main__.HomePageTest) ... ok
test_shopping_cart_status (__main__.HomePageTest) ... ok
test_vip_promo (__main__.HomePageTest) ... ok

-----
Ran 7 tests in 36.347s

OK
```

3.4 章节回顾

在本章，我们一起学习了许多重要的Selenium定位元素的方法。使用find_element_by方法，通过ID、name、class name、tag name、XPath、CSS选择器以及文本链接（或部分）去定位元素。

在后续设计测试时，可以更灵活地去运用这些定位的策略。这些知识为接下来的章节，如何调用Selenium API奠定了基础。

下一章，将会学习如何使用Selenium WebDriver的功能去与定位到的元素交互，以及模拟用户的操作（例如，在文本框输入、单击按钮、选择下拉菜单、调用JavaScript等操作）。

第4章 Selenium Python API 介绍

Web应用程序通过HTML表单的形式把数据发送到服务端。HTML表单中的输入元素包含文本框、复选框、单选框和提交按钮等。一个表单也可以包含下拉列表、文本域、插图和标签等元素。

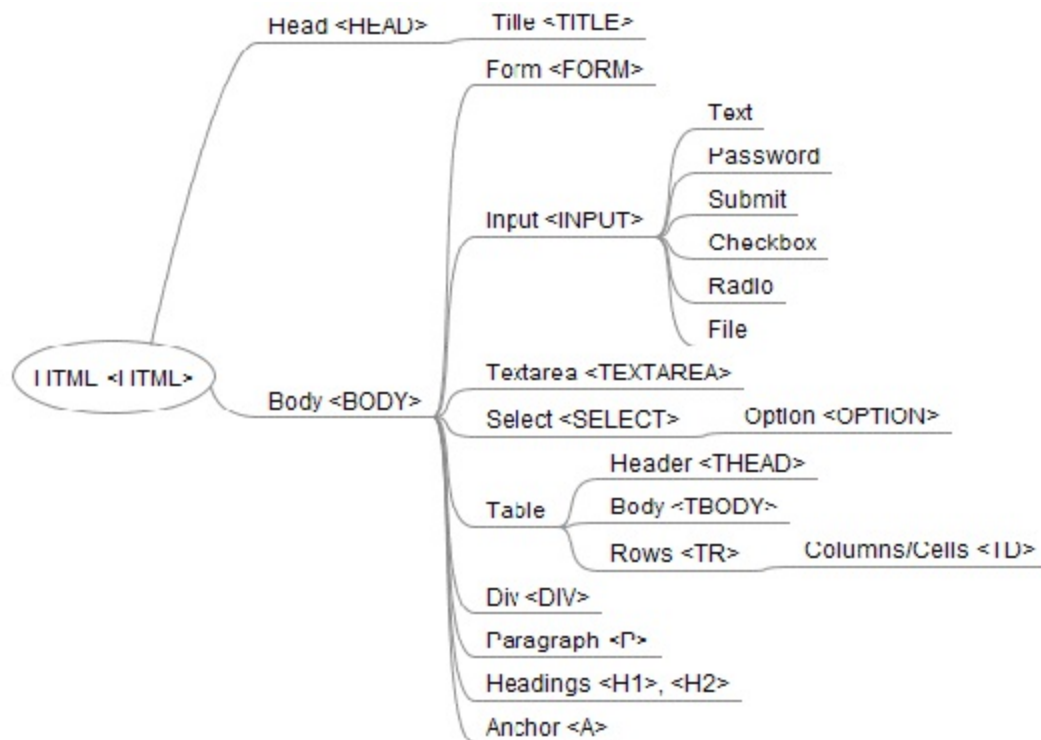
一个典型的Web应用程序从注册用户或搜索产品开始，往往需要填写很多的表单信息。表单是内嵌在HTML代码的<form>标签里的。标签中指定了提交数据的方法，可以使用GET和POST方法，输入到表单请求的地址就是我们要提交数据的服务器地址。

本章包含以下主题：

- 更多地了解WebDriver 和 WebElement这两个类；
- 使用WebDriver和WebElement的方法来实现包含与Web应用程序交互的测试；
- 使用Select类来实现下拉菜单和列表的自动化操作；
- 实现JavaScript 警告和浏览器导航栏的自动化。

4.1 HTML表单元素

HTML表单是由不同类型的元素组成的，如下图所示包含<form>、<input>、<button>和<label>等元素。Web应用开发者通过使用这些元素来实现数据的展示和接收用户的数据提交。开发人员通过定义这些元素来编写Web页面的HTML代码。然而作为终端用户，我们看到的这些元素是通过诸如文本框、标签、按钮、复选框和单选按钮的形式展现出来的。HTML代码对于终端用户来说是不可见的。



Selenium WebDriver为“实现通过与这些Web元素的交互的自动化来验证Web程序的功能正确性”提供了强大的支持。

4.2 WebDriver原理

WebDriver提供许多用来与浏览器交互的功能和设置。我们可以通过使用WebDriver的功能和一些方法来实现与浏览器窗口、警告、框架和弹出窗口的交互。它也提供了自动化操作浏览器导航栏、设置cookies、截屏等方便我们测试的特性。在后面的章节中，我们将依次阐述这些WebDriver的重要特性。本节的表格中包含了一些将在本书后面章节中使用到的非常重要的功能和方法。



在下面的网址可以看到完整的WebDriver的properties和方法列表。

http://selenium.googlecode.com/git/docs/api/py/webdriver_remote/selenium.webdriver.remote.webdriver.html#moduleselenium.webdriver.re

mote.webdriver

4.2.1 WebDriver功能

WebDriver通过下表的功能来操纵浏览器。

功能/属性	描 述	实 例
current_url	获取当前页面的URL地址	driver.current_url
current_window_handle	获取当前窗口的句柄	driver.current_window_handle
name	获取该实例底层的浏览器名称	driver.name
orientation	获取当前设备的方位	driver.orientation
page_source	获取当前页面的源代码	driver.page_source
title	获取当前页面的标题	driver.title
window_handles	获取当前session里所有窗口的句柄	driver.window_handles

4.2.2 WebDriver方法

WebDriver通过一些方法来实现与浏览器窗口、网页和页面元素的交互。下表是一些重要的方法。

方 法	描 述	参 数	实 例
back()	后退一步到当前会话的浏览器历史记录中最后一步操作前的页面		driver.back()
close()	关闭当前浏览器窗口		driver.close()
forward()	前进一步到当前会话的浏览器历史记录中前一步操作后的页面		driver.forward()
get(url)	访问目标URL并加载网页到当前的浏览器会话	URL是目标网页的网站地址	driver.get("http://www.googl
maximize_window()	最大化当前浏览器窗口		driver.maximize_window()
quit()	退出当前driver并且关闭所有的相关窗口		driver.quit()

refresh()	刷新当前页面		driver.refresh()
switch_to_active_element()	返回当前页面唯一焦点所在的元素或者元素体		driver.switch_to_active_element()
switch_to_alert()	把焦点切换至当前页面弹出的警告		driver.switch_to_alert()
switch_to_default_content()	切换焦点至默认框架内		driver.switch_to_default_content()
switch_to_frame(frame_reference)	通过索引、名称和网页元素将焦点切换到指定的框架，这种方法也适用于IFRAMES	frame_reference: 要切换的目标窗口的名称、整数类型的索引或者要切换的目标框架的网页元素	driver.switch_to_frame('frame_reference')
switch_to_window(window_name)	切换焦点到指定的窗口	window_name: 要切换的目标窗口的名称或者句柄	driver.switch_to_window('window_name')
implicitly_wait(time_to_wait)	超时设置等待目标元素被找到，或者目标指令执行完成。该方法在每个session只需要调用一次。 execute_async_script	time_to_wait: 等待时间（单位为秒）	

	的超时设置，请参阅set_script_timeout方法		
set_page_load_timeout(time_to_wait)	设置一个页面完全加载完成的超时等待时间	time_to_wait: 等待时间（单位为秒）	driver.set_page_load_timeou
set_script_timeout(time_to_wait)	设置脚本执行的超时时间，应该在execute_async_script抛出错误之前	time_to_wait: 等待时间（单位为秒）	driver.set_script_timeout(30)

4.3 WebElement接口

我们可以通过WebElement实现与网站页面上的元素的交互。这些元素包含文本框、文本域、按钮、单选框、多选框、表格、行、列和div等。

WebElement提供了一些功能、属性和方法来实现与网页元素的交互。本节的表格中将列出后面章节会用到的一些重要的功能和方法。如果想查看完整的功能和方法详情，请访问以下网站。

<http://selenium.googlecode.com/git/docs/api/py/webdriver.remote.webdriver.html#module-selenium.webdriver.remote.webdriver>

4.3.1 WebElement功能

下面是WebElement功能列表。

功能/属性	描 述	实 例

size	获取元素的大小	element.size
tag_name	获取元素的HTML标签名称	element.tag_name
text	获取元素的文本值	element.text

4.3.2 WebElement方法

下面是WebElement方法列表。

方 法	描 述	参 数	实 例
clear()	清除文本框或者文本域中的内容		element.clear()
click()	单击元素		element.click()
get_attribute(name)	获取元素的属性值	name: 元素的名称	element.get_attribute("value") 或者 element.get_attribute("maxlength")
is_displayed()	检查元素对于用户是否可见		element.is_displayed()
is_enabled()	检查元素是否可用		element.is_enabled()
	检查元素是否被		

is_selected()	选中。该方法应用于复选框和单选按钮		element.is_selected()
send_keys(*value)	模拟输入文本	value: 待输入的字符串	element.send_keys("foo")
submit()	用于提交表单。如果对一个元素应用此方法，将会提交该元素所属的表单		element.submit()
value_of_css_property(property_name)	获取CSS属性的值	property_name: CSS属性的名称	element.value_of_css_property("backgroundcolor")

4.4 操作表单、文本框、复选框、单选按钮

我们可以使用WebElement实现与各种HTML控件的自动化交互，例如在一个文本框输入文本、单击一个按钮、选择单选按钮或者复选框、获取元素的文本和属性值等。

在前面的章节可以看到WebElement提供的功能和方法。在本节中，我们将使用WebElement及其功能和方法实现在样例程序中创建账户功能的自动化。接下来我们创建一个测试脚本，来验证被测程序是否能正确创建一个新的账户。我们将按照下图来填写表单信息并且提交请求，系统收到请求后应该创建一个新的账户。

CREATE AN ACCOUNT

Please enter the following information to create your account.

First Name *

Last Name *

Email Address *

Password *

Confirm Password *

☐

Sign Up for Newsletter

[< Back](#)

REGISTER

正如在上图看到的，我们需要填写5个文本框并且选择一个复选框。

(1) 首先，创建一个新的测试类 RegisterNewUser，下面是实例代码。

```
from selenium import webdriver
import unittest
```

```
class RegisterNewUser(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com
/")
```

（2）添加一个测试方法

test_register_new_user(self) 到RegisterNewUser类中。

（3）为了打开登录页面，我们需要单击主页的登录链接。用于登录的代码如下。

```
def test_register_new_user(self):
    driver = self.driver

    # click on Log In link to open Login page
    driver.find_element_by_link_text("Log In").click()
```

4.4.1 检查元素是否启用或显示

当元素在屏幕上可见的时候（visible属性设置

为TRUE），调用is_displayed()方法返回为TRUE，反之就会返回FALSE。类似地，当元素是可用的时候，调用is_enabled()方法返回为TRUE，这时用户就可以执行点击和输入文本等操作。当元素是不可用的时候，该方法返回FALSE。

用户登录页面提供了使用已有账户登录和创建新用户的选项。我们可以通过调用is_displayed()方法和is_enabled()方法检查创建新账户按钮对于用户是否可见并且可用。添加下面的代码到测试类中。

```
# get the Create Account button
create_account_button = driver.find_element_by_xpath("//button[@title='Create an Account']")

# check Create Account button is displayed and enabled
self.assertTrue(create_account_button.is_displayed() and
                  create_account_button.is_enabled())
```

我们要测试创建账户功能，因此要单击创建账

户按钮，然后将会展示创建新账户的页面。我们可以通过检查WebDriver 的 title 属性来校验打开的页面是否符合预期结果，代码如下。

```
# click on Create Account button. This will display
# new account
create_account_button.click()

# check title
self.assertEqual("Create New Customer Account -
    Magento Commerce Demo Store", driver.title)
```

在创建新账户页面，可以通过调用 find_element_by_* 方法来查找定位所有的元素。

```
# get all the fields from Create an Account form
first_name = driver.find_element_by_id("firstname")
last_name = driver.find_element_by_id("lastname")
email_address = driver.find_element_by_id("email_addresses")
news_letter_subscription =
    driver.find_element_by_id("is_subscribed")
password = driver.find_element_by_id("password")
confirm_password = driver.find_element_by_id("confirmation")
submit_button =
    driver.find_element_by_xpath("//button[@title='Submit']")
```

4.4.2 获取元素对应的值

`get_attribute()`方法可以用来获取元素的属性值。例如，单个测试是用来验证输入姓和名字的文本框的最大字符限制是255，字符限制就是通过`maxlength`属性来实现的，如下代码所示设置值为255。

```
<input type="text" id="firstname" name="firstname" value=""  
      title="First Name" maxlength="255" class="input-text  
      required-entry">
```

我们可以通过调用`get_attribute()`方法来校验`maxlength`属性是否正确。

(1) 需要把属性名称作为参数传递给`get_attribute()`方法。

```
# check maxlength of first name and last name textbox  
self.assertEqual("255", first_name.get_attribute("maxle  
ngth"))  
self.assertEqual("255", last_name.get_attribute("maxlen  
gth"))
```

(2) 添加以下代码到测试脚本中，以确保所有的字段对于用户都是可见和可用的。

```
# check all fields are enabled
self.assertTrue(first_name.is_enabled() and last_name.is_enabled()
                 and email_address.is_enabled() and newsletter_subscription.is_enabled()
                 and password.is_enabled() and confirm_password.is_enabled()
                 and submit_button.is_enabled())
```

4.4.3 is_selected()方法

is_selected() 方法是针对单选按钮和复选框的。我们可以通过调用该方法来得知一个单选按钮或复选框是否被选中。

单选按钮或复选框可以通过WebElement 的 click() 方法来执行点击操作，从而选中该元素。如下面的例子，检查Sign UP for Newsletter 复选框是否默认为不被选中的，示例代码如下。

```
# check Sign Up for Newsletter is unchecked
```



```
self.assertFalse(news_letter_subscription.is_selected())
```

4.4.4 clear()与send_keys()方法

clear() 和 send_keys()方法适用于文本框和文本域，分别用于清除元素的文本内容和模拟用户操作键盘来输入文本信息。待输入的文本作为 send_keys() 方法的参数。

(1) 添加下面的代码，通过send_keys() 方法来给对应的字段填写值。

```
# fill out all the fields
first_name.send_keys("Test")
last_name.send_keys("User1")
news_letter_subscription.click()
email_address.send_keys("TestUser_150214_2200@example.com")
password.send_keys("tester")
confirm_password.send_keys("tester")
```

(2) 最终通过校验欢迎信息来检查用户是否创建成功。

我们可以通过text 属性来获取元素的文本内容。

```
# check new user is registered
self.assertEqual("Hello, Test User1!", driver.find_element_by_css_selector("p.hello > strong").text)
self.assertTrue(driver.find_element_by_link_text("LogOut").is_displayed())
```

(3) 下面是创建一个账户功能的完整测试。运行这个测试脚本将看到在 Create An Account 页面的所有操作。

```
from selenium import webdriver
from time import gmtime, strftime
import unittest

class RegisterNewUser(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com/")

    def test_register_new_user(self):
        driver = self.driver
```

```

        # click on Log In link to open Login page
        driver.find_element_by_link_text("ACCOUNT").click()

    def create_account():
        # get the Create Account button
        create_account_button = \
            driver.find_element_by_link_text("CREATE AN ACCOUNT")

        # check Create Account button is displayed
        # and enabled
        self.assertTrue(create_account_button.is_displayed() and
            create_account_button.is_enabled())

        # click on Create Account button. This will
        # display new account
        create_account_button.click()

        # check title
        self.assertEqual("Create New Customer Account",
            driver.title)

        # get all the fields from Create an Account form
        first_name = driver.find_element_by_id("firstname")
        last_name = driver.find_element_by_id("lastname")
        email_address = driver.find_element_by_id("email_address")
        password = driver.find_element_by_id("password")
        confirm_password = driver.find_element_by_id("confirmation")

```

```

        news_letter_subscription = driver.find_element_b
y_id("is_subscribed")
        submit_button = driver.\find_element_by_xpath ("
//button[@title='Register']")

        # check maxlength of first name and
        # last name textbox
        self.assertEqual("255", first_name.get_attribute
("maxlength"))
        self.assertEqual("255", last_name.get_attribute(
"maxlength"))

        # check all fields are enabled
self.assertTrue(first_name.is_enabled()
        and last_name.is_enabled()
        and email_address.is_enabled() and
        news_letter_subscription.is_enabled() and
        password.is_enabled() and
        confirm_password.is_enabled()
        and submit_button.is_enabled())

        # check Sign Up for Newsletter is unchecked
self.assertFalse(news_letter_subscription. is_se
lected())

        user_name = "user_" + strftime ("%Y%m%d%H%M%S",
gmtime())

        # fill out all the fields
first_name.send_keys("Test")
last_name.send_keys(user_name)
news_letter_subscription.click()
email_address.send_keys(user_name + "@example.co
m")
password.send_keys("tester")

```

```
confirm_password.send_keys("tester")

# click Submit button to submit the form
submit_button.click()

# check new user is registered
self.assertEqual("Hello, Test " + user_name + "!
",
    driver.find_element_by_css_selector("p.hello
>strong").text)
    driver.find_element_by_link_text("ACCOUNT").clik
k()
    self.assertTrue(driver.find_element_by_link_text
("Log Out").is_displayed())

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

4.5 操作下拉菜单

Selenium WebDriver提供了特定的Select类实现与网页上的列表和下拉菜单的交互。例如下面的样例程序，可以看到一个为店铺选择语言的下拉菜单。



下拉菜单和列表是通过HTML的<select> 元素实现的。选择项是通过<select>中的<option>元素实现的，如下HTML代码。

```
<select id="select-language" title="Your Language"
  onchange="window.location.href=this.value">
  <option value="http://demo.magentocommerce.com/?
    __store=default&__from_store=default"
```

```
selected="selected">English</option>
<option value="http://demo.magentocommerce.com/?
    __store=french&__from_store=default">French</
option>
<option value="http://demo.magentocommerce.com/?
    __store=german&__from_store=default">German</
option>
</select>
```

每个<option> 元素都有属性值和文本内容，是用户可见的。例如，在下面的代码中，<option>设置的是店铺的URL，后面参数设置的是语言种类，这里是French。

```
<option value="http://demo.magentocommerce.com/customer
/
account/create/?__store=french&
__from_store=default">French</option>
```

4.5.1 Select原理

Select 类是Selenium的一个特定的类，用于与下拉菜单和列表交互。它提供了丰富的功能和方法来实现与用户交互。

下面两小节的表格列出来Select类中所有的功能和方法。你也可以在下面网址获取类似信息。

<http://selenium.googlecode.com/git/docs/api/py/webdriver/selenium.webdriver.support.select>

4.5.2 Select功能

Select类实现的功能见下表。

功能/属性	描 述	实 例
all_selected_options	获取下拉菜单和列表中被选中的所有选项内容	select_element.all_selected_options
first_selected_option	获取下拉菜单和列表的第一个选项 / 当前选择项	select_element.first_selected_option
options	获取下拉菜单和列表的所有选项	select_element.options

4.5.3 Select方法

Select类实现的方法见下表。

功能/属性	描 述	参 数	实 例
deselect_all()	清除多选下拉菜单和列表的所有选择项		select_element.deselect_all()
deselect_by_index(index)	根据索引清除下拉菜单和列表的选择项	index: 要清除的目标选择项的索引	select_element.deselect_by_index(1)
deselect_by_value(value)	清除所有选项值和给定参数匹配的下拉菜单和列表的选择项	value: 要清除的目标选择项的value属性	select_element.deselect_by_value("foo")
deselect_by_visible_text(text)	清除所有展示的文本和给定参数匹配的下拉菜单	text: 要清除的目标选择项的	select_element.deselect_by_visible_text("bar")

	和列表的选择项	文本值	
<code>select_by_index(index)</code>	根据索引选择下拉菜单和列表的选择项	index: 要选择的 目标选择项 的索引	<code>select_element.select_by_index(1)</code>
<code>select_by_value(value)</code>	选择所有选项值和给定参数匹配的下拉菜单和列表的选择项	value: 要选择的 目标选择项 的value 属性	<code>select_element.select_by_value("foo")</code>
<code>select_by_visible_text(text)</code>	选择所有展示的文本和给定参数匹配的下拉菜单和列表的选择项	text: 要选择的 目标选择项的 文本值	<code>select_element.select_by_visible_text("bar")</code>

让我们进一步探究这些功能和方法，我们回到刚才被测网站的语言选择功能。我们将为前面章节创建好的主页面的测试类添加一个新的测试用例。这个测试用例用来验证是否有8种语言可供用户选择。我们将首先使用options 属性来验证选项的个数是否和预期结果一致，然后通过获取每个选项的文本与预期的选项列表相比较，从而校验是否一致，代码如下所示。

```
def test_language_options(self):
    # list of expected values in Language dropdown
    exp_options = ["ENGLISH", "FRENCH", "GERMAN"]

    # empty list for capturing actual options displayed
    # in the dropdown
    act_options = []

    # get the Your language dropdown as instance of Select class
    select_language = \
        Select(self.driver.find_element_by_id("select-language"))

    # check number of options in dropdown
    self.assertEqual(2, len(select_language.options))

    # get options in a list
```

```
for option in select_language.options:
    act_options.append(option.text)

# check expected options list with actual options list
self.assertEqual(exp_options, act_options)

# check default selected option is English
self.assertEqual("ENGLISH", select_language.first_selected_option.text)

# select an option using select_by_visible_text
select_language.select_by_visible_text("German")

# check store is now German
self.assertTrue("store=german" in self.driver.current_url)

# changing language will refresh the page,
# we need to get find language dropdown once again
select_language = \
    Select(self.driver.find_element_by_id("select-language"))
select_language.select_by_index(0)
```

`options`属性返回一个下拉选项和列表里的所有`<option>`元素。选项列表里的每个选项都是一个`WebElement`类的实例。

我们也可以通过用`first_selected_option`属性来

校验默认/当前选择项是否正确。



`all_selected_options` 属性是用来测试多选的下拉选项和列表的。

最后，我们用下面的代码来实现：选择一个语言选项，然后校验保存的**URL**是否能够随着语言选项的改变而正确地变化。

```
# select an option using select_by_visible text
select_language.select_by_visible_text("German")

# check store is now German
self.assertTrue("store=german" in self.driver.current_url)
```

一个或多个选项可以基于索引来选择（该选项在列表中的位置），也可以根据属性值或者文本值来选择。`select`类提供了很多`select_`方法来选择选

项。在上面这个例子中，我们使用 `select_by_visible_text()` 方法来选择选项。反之，我们也可以用各种 `deselect_` 方法来取消选择。

4.6 操作警告和弹出框

开发人员使用JavaScript 警告或者模态对话框来提示校验错误信息、报警信息、执行操作后的返回信息，甚至用来接收输入值等。本节我们将了解如何使用Selenium来操控警告和弹出框。

4.6.1 Alert 原理

Selenium WebDriver 通过Alert 类来操控JavaScript 警告。Alert 包含的方法有接受、驳回、输入和获取警告的文本。

4.6.2 Alert功能

Alert 实现了下表的功能。

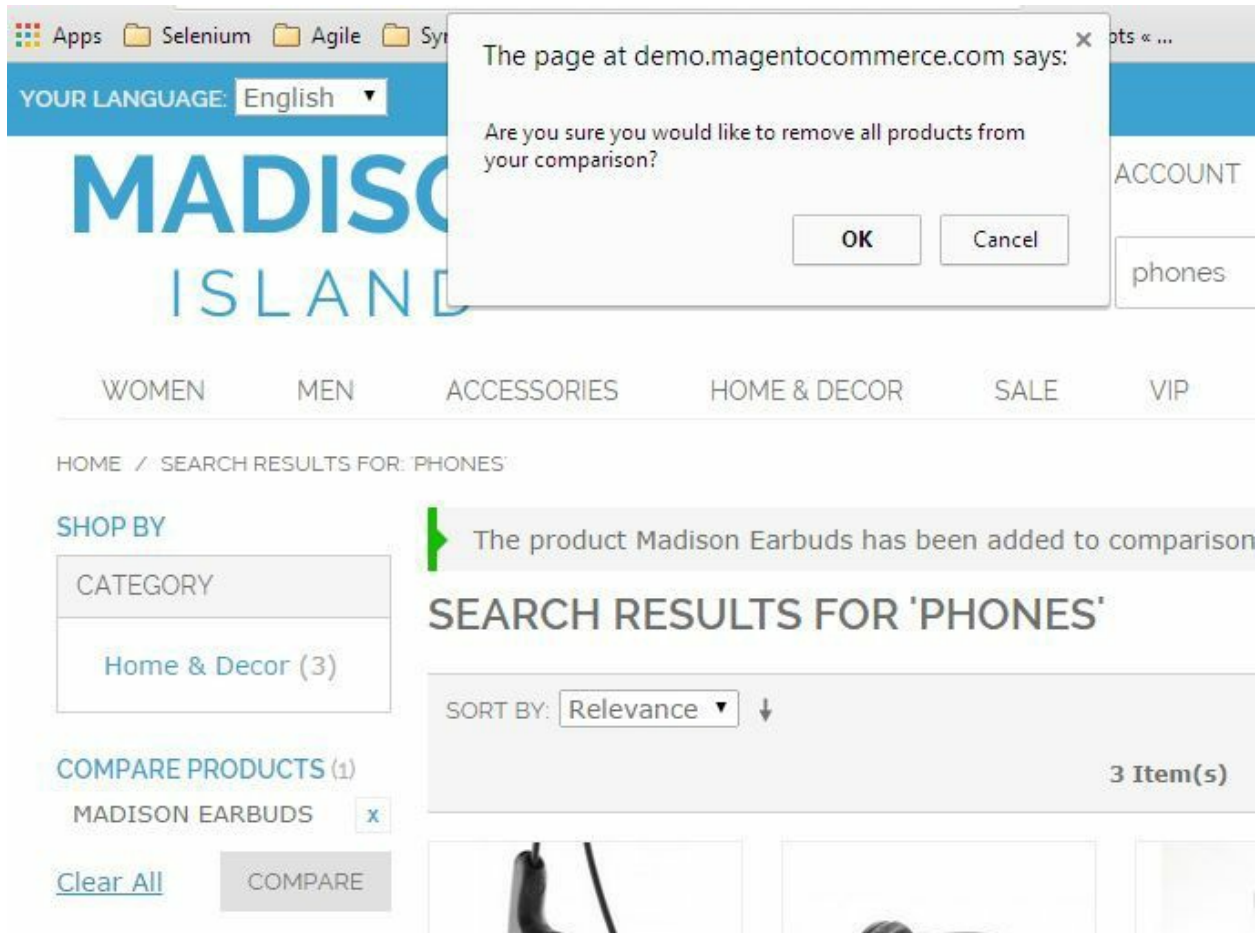
功能/属性	描 述	实 例
text	获取警告窗口的文本	alert.text

4.6.3 Alert方法

Alert实现了下表的方法。

方 法	描 述	参 数	实 例
accept()	接受JavaScript 警告信息，单击OK 按钮		alert.accept()
dismiss()	驳回JavaScript 警告信息，单击取消按钮		alert.dismiss()
send_keys(*value)	模拟给元素输入信息	value: 待输入目标字段的字符串	alert.send_keys("foo")

在样例程序中，可以看到使用Alert 通知或告警。用户先添加产品进行比较，然后移除一个或多个产品时，被测程序将会显示一个如下图这样的告警信息。



我们将设计一个测试来验证单击COMPARE PRODUCTS（产品比较）功能中的Clear All链接时，是否会弹出警告提醒用户。

创建一个新的测试类CompareProducts，并添加测试场景的代码，搜索并添加一个产品到比较列表中，代码如下。

```
from selenium import webdriver
import unittest

class CompareProducts(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()
        self.driver.get("http://demo.magentocommerce.com
/")

    def test_compare_products_removal_alert(self):
        # get the search textbox
        search_field = self.driver.find_element_by_name(
"q")
        search_field.clear()

        # enter search keyword and submit
        search_field.send_keys("phones")
        search_field.submit()

        # click the Add to compare link
        self.driver.\
            find_element_by_link_text("Add to Compare").
click()
```

当单击Add to Compare 链接将一个产品添加到比较列表时，将会看到一个产品添加到COMPARE PRODUCTS 下面。这个时候还可以添加其他的产
品到比较列表。如果想从比较列表移除所有的产

品，可以在COMPARE PRODUCTS 模块单击Clear All 链接。这个时候可以看到一个警告提示“是否确认移除所有的产品”。我们可以通过Alert来操控这个警告。调用WebDriver 的Switch_to_alert() 方法可以返回一个Alert的实例。我们可以利用这个Alert实例来获取警告信息，并通过单击OK按钮来接受这个警告信息，或者通过单击Cancel 按钮来拒绝这个警告。添加下面的代码到测试脚本中，这部分代码用来读取并且校验警告信息是否正确，然后通过调用accept() 方法来接受警告。

```
# click on Remove this item link, this will display
# an alert to the user
self.driver.find_element_by_link_text("Clear All").
click()

# switch to the alert
alert = self.driver.switch_to_alert()

# get the text from alert
alert_text = alert.text

# check alert text
self.assertEqual("Are you sure you would like to
remove all products from your comparison?", alert_
```

```
text)

    # click on Ok button
    alert.accept()

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":
    unittest.main()
```

4.6.4 浏览器自动化处理

通过单击浏览器工具栏上的后退、前进、刷新 / 重新加载按钮，可以实现访问历史页面、刷新当前页面等操作。Selenium WebDriver API 提供了很多操控这些按钮的方法，我们可以使用这些方法来验证浏览器的行为。WebDriver 类提供了以下方法来操控浏览器的后退、前进和刷新等操作。

方 法	描 述	参 数	实 例
back()	后退到浏览器当前会话的历史记录中的前一步操作	无	driver.back()
	向前一步到浏览器当前会话的历史记录中的后一步操		

forward()	作	无	driver.forward()
refresh()	刷新浏览器中的当前页面	无	driver.refresh()

下面的例子是通过浏览器API操控浏览器历史记录并验证程序的状态。

```
import unittest
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions

class NavigationTest(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Chrome()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://www.google.com")

    def testBrowserNavigation(self):
        driver = self.driver
        # get the search textbox
        search_field = driver.find_element_by_name("q")
        search_field.clear()

        # enter search keyword and submit
        search_field.send_keys("selenium webdriver")
```

```
search_field.submit()

se_wd_link = driver.find_element_by_link_text("Selenium WebDriver")
se_wd_link.click()
self.assertEqual("Selenium WebDriver", driver.title)

driver.back()
self.assertTrue(WebDriverWait(self.driver, 10)
    .until(expected_conditions.title_is
        ("selenium webdriver - Google Search")))

driver.forward()
self.assertTrue(WebDriverWait(self.driver, 10)
    .until(expected_conditions.title_is
        ("Selenium WebDriver")))

driver.refresh()
self.assertTrue(WebDriverWait(self.driver, 10)
    .until(expected_conditions.title_is
        ("Selenium WebDriver")))

def tearDown(self):
    # close the browser window
    self.driver.quit()

if __name__ == '__main__':
    unittest.main()
```

4.7 章节回顾

本章介绍了Selenium WebDriver API 与页面各种元素的交互实现。Selenium WebDriver API 提供了不同的类、功能和方法来模拟用户的动作，从而校验应用程序的状态。这些方法能够自动化操控的元素有文本框、按钮、复选框和下拉列表等。

同时，我们还设计了一些处理警告的测试，学习了操控浏览器的方法，并且测试了浏览器在不同页面之间的跳转。

在下一章，我们将进一步学习Selenium API如何来处理同步机制，这些内容能够帮助我们构建更加稳定的测试。

第5章 元素等待机制

能否构建健壮和可靠的测试是UI自动化测试能否成功的关键因素之一。然而当一个测试接着一个测试执行的时候，常常会遇到各种不同的状况。当使用脚本定位元素或去验证程序的运行状态时，有时候会发现找不到元素，这可能是由于突然的资源受限或网络延迟引起的响应速度太慢所导致，这时测试报告就会返回测试失败的结果。我们需要在测试脚本中引入延时机制，来使脚本的运行速度与程序的响应速度相匹配。换句话说，我们需要使脚本和程序的响应能够同步。WebDriver为这种同步提供了隐式等待和显式等待两种机制。

本章包含以下主题：

- 如何使用隐式等待或显式等待？

- 什么情况下使用隐式等待或显式等待？
- 使用预期等待条件；
- 创建自定义的等待条件。

5.1 隐式等待

隐式等待为WebDriver中的完整的一个测试用例或者一组测试的同步，提供了通用的方法。隐式等待对于解决由于网络延迟或利用Ajax动态加载元素所导致的程序响应时间不一致，是非常有效的。

当设置了隐式等待时间后，WebDriver会在一定的时间内持续检测和搜寻DOM，以便于查找一个或多个不是立即加载成功并可用的元素。一般情况下，隐式等待的默认超时时间设置为0。

一旦设置，隐式等待时间就会作用于这个WebDriver实例的整个生命周期或者一次完整测试的执行期间，并且WebDriver会使其对所有测试步骤中包含整个页面的元素的查找时都有效，除非把默认超时时间设置回0。

WebDriver类提供了implicitly_wait()方法来配置超时时间。本书在第2章已经创建了SearchProductTest测试类，是基于unittest写的测试。我们将基于这个类进行修改，在setUp()方法中加入隐式等待时间并且设置为10秒，代码如下面的例子所示。当一个测试用例执行的时候，WebDirver在找不到一个元素的时候，将会等待10秒。当达到10秒超时时间后，将会抛出一个NoSuchElementException 的异常。

```
import unittest
from selenium import webdriver

class SearchProductTest(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com/")

    def test_search_by_category(self):
        # get the search textbox
```

```

        self.search_field = self.driver.find_element_by_
name("q")
        self.search_field.clear()

        # enter search keyword and submit
        self.search_field.send_keys("phones")
        self.search_field.submit()

        # get all the anchor elements which have product
names
        # displayed currently on result page using
        # find_elements_by_xpath method
        products = self.driver\
            .find_elements_by_xpath
            ("//h2[@class='product-name']/a")

        # check count of products shown in results
        self.assertEqual(2, len(products))

    def tearDown(self):
        # close the browser window
        self.driver.quit()

if __name__ == '__main__':
    unittest.main(verbosity=2)

```



应尽量避免在测试中隐式等待与显式等待混合使用，来处理同步问题。相比隐式等待，显式等待能提供更好的可控性。



5.2 显式等待

显式等待是WebDriver中用于同步测试的另外一种等待机制。显式等待比隐式等待具备更好的操控性。与隐式等待不同，我们可以为脚本设置一些预置或定制化的条件，等待条件满足后再进行下一步测试。

显式等待可以只作用于仅有同步需求的测试用例。WebDriver提供了WebDriverWait类和expected_conditions类来实现显式等待。

expected_conditions类提供了一些预置条件，来作为测试脚本进行下一步测试的判断依据。让我们创建一个包含显式等待的简单的测试，条件是等待一个元素可见，代码如下。

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
```

```

from selenium.webdriver.support import expected_conditions
import unittest

class ExplicitWaitTests(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.get("http://demo.magentocommerce.com/")

    def test_account_link(self):
        WebDriverWait(self.driver, 10)\
            .until(lambda s: s.find_element_by_id("select-
            language").get_attribute("length") == "3")
        account = WebDriverWait(self.driver, 10)\
            .until(expected_conditions.
            visibility_of_element_located
            ((By.LINK_TEXT, "ACCOUNT")))
        account.click()

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":
    unittest.main(verbosity=2)

```

在上面的测试中，显式等待条件是等到Log In 链接在DOM中可见。

使用visibility_of_element_located方法来判断预

期条件是否满足。该条件判断方法需要设置符合要求的定位策略和位置详细信息。脚本将一直查找目标元素是否可见，直到达到最大等待时间10秒。一旦根据指定的定位器找到了元素，预期条件判定方法将会把定位到的元素返回给测试脚本。

如果在设定的超时时间内，仍然没有通过定位器找到可见的目标元素，将会抛出 `TimeoutException` 异常。

5.3 expected_conditions类

下表是expected_conditions类支持的在执行网页浏览器自动化操作时常常用到的一些通用的等待条件。

预 期 条 件	描 述	参 数	
element_to_be_clickable(locator)	等待通过定位器查找的元素可见并且可用，以便确定元素是可点击的。 此方法返回定位到的元素	locator: 一组 (by,locator)	WebDriverWait(self.driver, 10).until(expected_conditions.e
element_to_be_selected(element)	等待直到指定的元素被选中	element: 是个 WebElement	subscription = self.driver.find_e WebDriverWait(self.driver, 10) element_to_be_selected(subscri
invisibility_of_element_located(locator)	等待一个元素在DOM中不可见或不存在	locator: 一组 (by,locator)	WebDriverWait(self.driver, 10) invisibility_of_element_located

presence_of_all_elements_located(locator)	等待直到至少有一个定位器查找匹配到的目标元素出现在网页中。 该方法返回定位到的一组 WebElement	locator: 一组 (by,locator)	WebDriverWait(self.driver, 10) until(expected_conditions.presence_of_element_located(locator,by,locator)))
presence_of_element_located(locator)	等待直到定位器查找匹配到的目标元素出现在网页中或可以在DOM中找到。 该方法返回一个被定位到的元素	locator: 一组 (by,locator)	WebDriverWait(self.driver, 10) until(expected_conditions.presence_of_element_located(locator,by,locator)))
text_to_be_present_in_element(locator,text_)	等待直到元素能被定位到并且带有相应的文本信息	locator: 一组 (by,locator) text:需要被校验的文本内容	WebDriverWait(self.driver,10). until(expected_conditions.text_to_be_present_in_element(locator,by,locator),text_))
title_contains(title)	等待网页标题包含指定的大小写敏感的字符串。 该方法在匹配成功时返回	title: 被校验的包含在标题中的字符串	WebDriverWait(self.driver, 10) until(expected_conditions.title_contains(title))

	回True，否则返回False		
title_is(title)	等待网页标题与预期的标题相一致。 该方法在匹配成功时返回True，否则返回False	title:网页的标题	WebDriverWait(self.driver, 10) CustomerAccount -MagentoCo
visibility_of(element)	等待直到元素出现在DOM中，是可见的，并且宽和高都大于0。一旦其变成可见的，该方法将返回（同一个）WebElement	element : 目标 WebElement	first_name = self.driver.find_ele 10). until(expected_conditions.v
visibility_of_element_located(locator)	等待直到根据定位器查找的目标元素出现在DOM中，是可见的，并且宽和高都大于0。一旦其变成可见的，该方法将返回	locator: 一组 (by,locator)	WebDriverWait(self.driver, 10) until(expected_conditions.visibi

	WebElement		
--	------------	--	--

在下面的网址可看到预期条件判断的完整列表：

http://selenium.googlecode.com/git/docs/api/py/webdriver/webdriver.support.expected_conditions.html#module-selenium.webdriver.support.expected_conditions。

在下面的章节中，让我们通过几个例子来了解更多的预期条件判断。

5.3.1 判断某个元素是否存在

正如在前面章节看到的，`expected_conditions`类提供了各种各样的预期等待条件，我们可以在脚本中实现。在下面的例子里，我们将等待一个元素变成可用或可点击。我们可以在Ajax 应用较多的程序中使用这个预期等待条件，这样表单中一个字段是否可用取决于表单中别的字段或过滤器。该例子

中，我们单击Log In链接，然后等待Create an Account按钮变成可点击的，这些元素都在登录页面。最后我们单击Create an Account按钮，等待下一个页面加载完成并显示出来。

```
def test_create_new_customer(self):
    # click on Log In link to open Login page
    self.driver.find_element_by_link_text("ACCOUNT").click()

    # wait for My Account link in Menu
    my_account = WebDriverWait(self.driver, 10)\
        .until(expected_conditions.visibility_of_element_
        _located((By.LINK_TEXT, "My Account"))))
    my_account.click()

    # get the Create Account button
    create_account_button = WebDriverWait(self.driver,
    10)\
        .until(expected_conditions.element_to_be_clickable((By.LINK_
        TEXT, "CREATE AN ACCOUNT"))))

    # click on Create Account button. This will display
    ed new account
    create_account_button.click()
    WebDriverWait(self.driver, 10)\
        .until(expected_conditions.title_contains("Creat
    e New Customer Account"))
```

我们等待并检查一个元素是否可用，可以用 `element_to_be_clickable` 预期条件。该方法需要指定定位策略或具体定位的位置。当目标元素变成可点击或者可用的时候，该方法返回定位到的目标元素给测试脚本。

前面的测试也介绍了通过检测标题是否含有指定的文本内容，来确定创建新用户页面是否加载成功。我们使用 `title_contains` 预期条件来检测，以确保指定的字符串能够与预期网页标题的子字符串相匹配。

5.3.2 判断是否存在Alerts

我们也可以将显式等待应用于警告和页面框架中。例如，一个复杂的JavaScript处理过程或后端处理过程需要花费较多的时间把警告反馈给用户，这时可以用 `alert_is_present` 这个预期判断条件来实现，代码如下。

```

from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions
import unittest

class CompareProducts(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.get("http://demo.magentocommerce.com/")

    def test_compare_products_removal_alert(self):
        # get the search textbox
        search_field = self.driver.find_element_by_name("q")
        search_field.clear()

        # enter search keyword and submit
        search_field.send_keys("phones")
        search_field.submit()

        # click the Add to compare link
        self.driver.\
            find_element_by_link_text("Add to Compare").\
click()

        # wait for Clear All link to be visible
        clear_all_link = WebDriverWait(self.driver, 10)\
            .until(expected_conditions.visibility_of_element_
                located((By.LINK_TEXT, "Clear All")))

        # click on Clear All link,

```

```
# this will display an alert to the user
clear_all_link.click()

# wait for the alert to present
alert = WebDriverWait(self.driver, 10)\
    .until(expected_conditions.alert_is_present(
))

# get the text from alert
alert_text = alert.text

# check alert text
self.assertEqual("Are you sure you would like
to remove all products from your comparison?", alert_
text)

# click on Ok button
alert.accept()

def tearDown(self):
    self.driver.quit()

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

上述的测试脚本，是验证从产品比较列表中移除所有的产品这个功能。当用户移除一个产品的时候，会收到是否确认的警告。Alert_is_present 预期判定条件就可以用来检测警告窗口是否出现，并且把警告窗口返回给脚本，以进行后续的动作。该脚

本将会等待10秒的时间来检测警告窗口是否出现，
如果没有出现就抛出异常。

5.4 预期条件判断的实践

正如在前面章节所了解到的，`expected_conditions` 类提供了多种定义好的预期等待条件。我们也可以通过 `WebDriverWait` 来自定义预期等待条件。当没有合适的预期等待条件可用的时候，自定义的预期等待条件也是非常有效的。

让我们来修改一个前面章节中创建好的测试脚本，实现一个自定义的预期条件判断，来检测下拉列表中可选项的数量。

```
def testLoginLink(self):
    WebDriverWait(self.driver, 10).until
        (lambda s: s.find_element_by_id
         ("select-language").get_attribute("length") == "3")

    login_link = WebDriverWait
        (self.driver, 10).until(expected_conditions.
        visibility_of_element_located((By.LINK_TEXT, "Log
In")))
    login_link.click();
```

我们可以使用Python的lambda表达式，并且基于WebDriverWait来实现自定义的预期条件判断。上面的例子中，脚本将会等待10秒，直到Select Language下拉列表中有8个可选项。当下拉列表是通过Ajax调用来实现，并且脚本需要等待下拉列表中的所有选项都是可选择时，该预期条件判断是非常有用的。

5.5 章节回顾

在本章中，我们认识到元素等待机制对于构建高度稳定可靠的测试来说是必不可少的。我们学习了隐式等待，并且通过例子了解到了如何应用隐式等待作为通用的等待机制。显式等待可以提供更灵活的方式来同步进行测试。`expected_conditions`类提供了多种内置的预期等待判定条件，我们在例子中也实践了一部分。

`WebDriverWait`类提供了更加强大的自定义预期等待判定功能，超出了`expected_conditions`。我们在下拉列表的例子中就实现了自定义的预期等待判定。

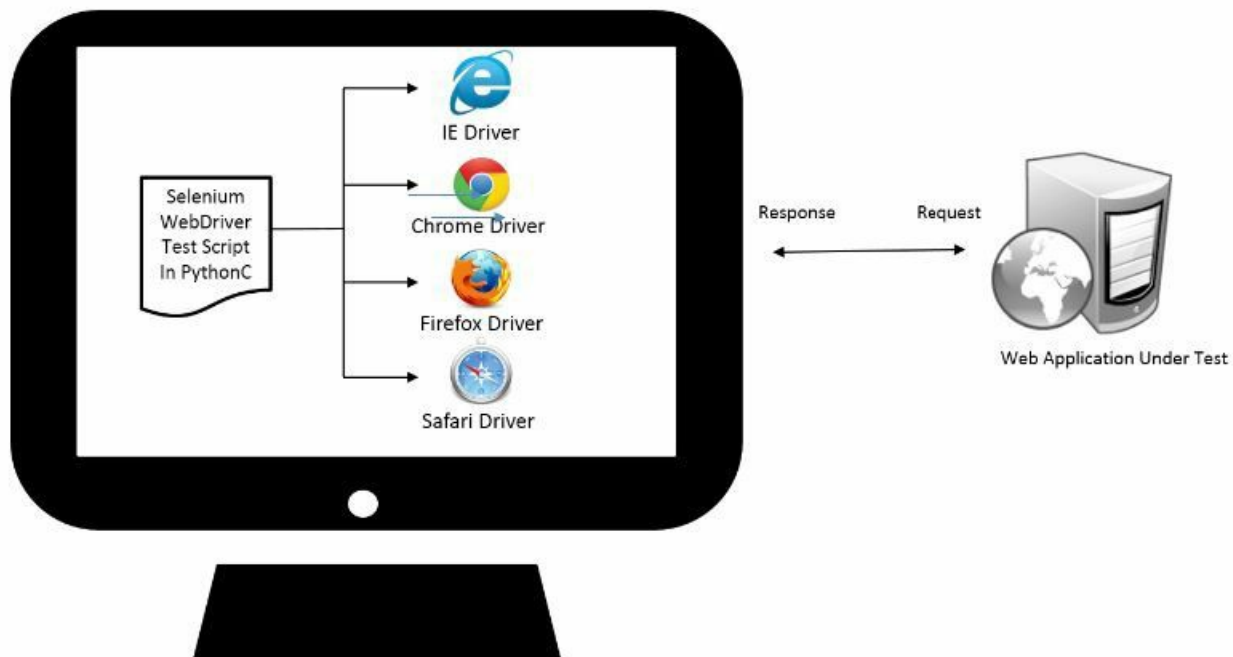
在下面的章节中，将会讲述如何通过使用`RemoteWebDriver`和`Selenium Server`使测试脚本在远程机器上执行，并且通过`Selenium Grid`实现脚本的

并行执行，进而实现跨浏览器自动化测试。

第6章 跨浏览器测试

Selenium支持由多种浏览器和操作系统组合的跨浏览器测试。该特性通过在不同浏览器和操作系统的组合场景下执行测试，来验证Web程序的跨浏览器兼容性，从而确保用户在他们喜好选择的浏览器和操作系统上使用程序时不会遇到问题。

Selenium WebDriver支持在远程机器上执行测试，并且能够把测试分发到安装有不同浏览器和操作系统的远程机器或者云端执行。到目前为止，我们已经学习了在安装各种浏览器驱动的本地计算机上如何创建和执行测试，如下图所示。



本章将学习如何在远程机器上执行测试，并且学习如何在由不同浏览器和操作系统组合成的分布式架构中的远程机器上批量执行跨浏览器测试。这种执行跨浏览器测试的实现方式将会节省大量的时间。

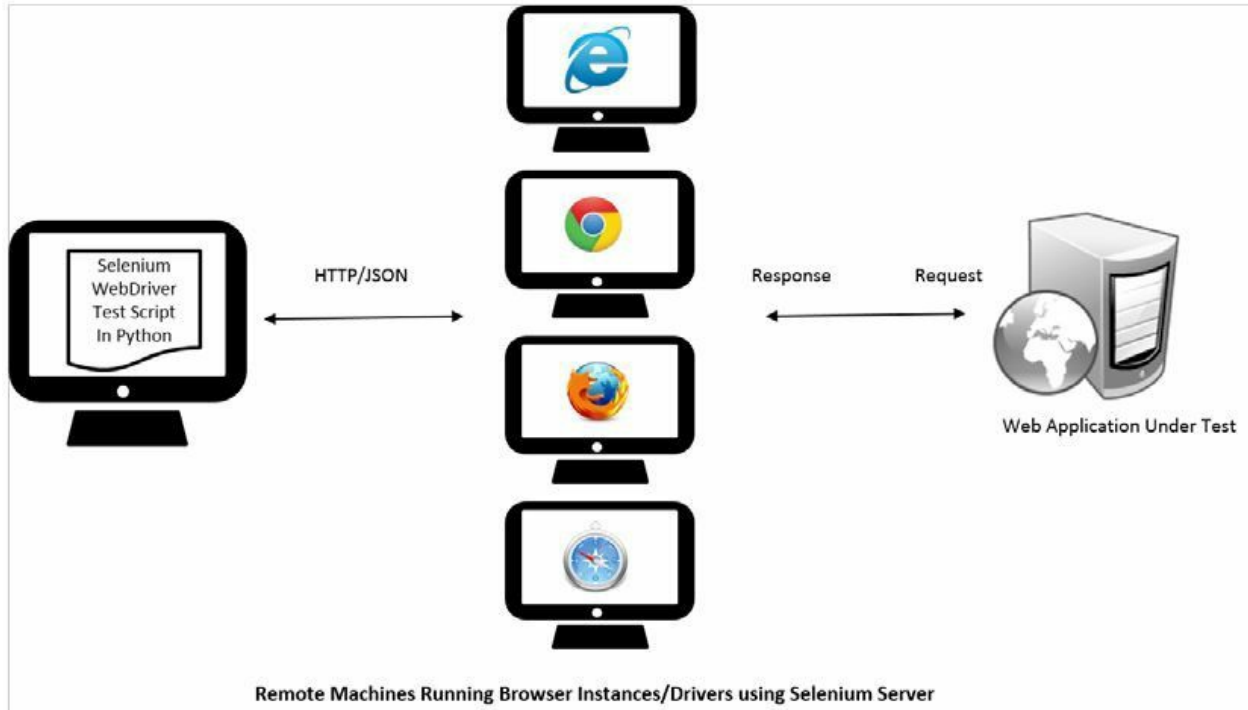
本章包含以下主题：

- Selenium Standalone Server 的下载和使用；
- 如何使用Remote 类来实现在Selenium Standalone Server 上执行测试；

- 在Selenium Standalone Server上执行测试;
- 为Selenium Standalone Server添加节点, 从而为分布式执行创建一个Grid;
- 在安装有多浏览器和操作系统组合的Grid上执行测试;
- 通过Sauce Labs 和BrowserStack在云端执行测试。

6.1 Selenium Standalone Server

Selenium Standalone Server 是使Selenium具备在远程机器上执行测试能力的一个重要组件。我们需要通过使用RemoteWebDriver类来连接到Selenium Standalone Server，从而实现在远程机器上执行测试。RemoteWebDriver类通过特定的端口监听Selenium根据测试脚本所下达的命令。根据RemoteWebDriver类提供的配置选项，Selenium Server可以选择启动的浏览器类型并且发送命令给浏览器。它几乎支持所有的浏览器，并且还可以基于Appium来实现对移动平台的支持。下面是Selenium Server在配置了不同类型浏览器的远程机器上执行测试的架构图。



6.1.1 下载Selenium Standalone Server

Selenium Standalone Server是以JAR包的形式下载，可以从[http://docs.seleniumhq.org/ download/](http://docs.seleniumhq.org/download/)页面的Selenium Server（原来的Selenium RC Server）章节找到。写这本书时，Selenium Server可下载版本是2.41.0。你可以轻松地将Selenium Standalone Server的JAR包文件复制到远程机器上并启动服务。



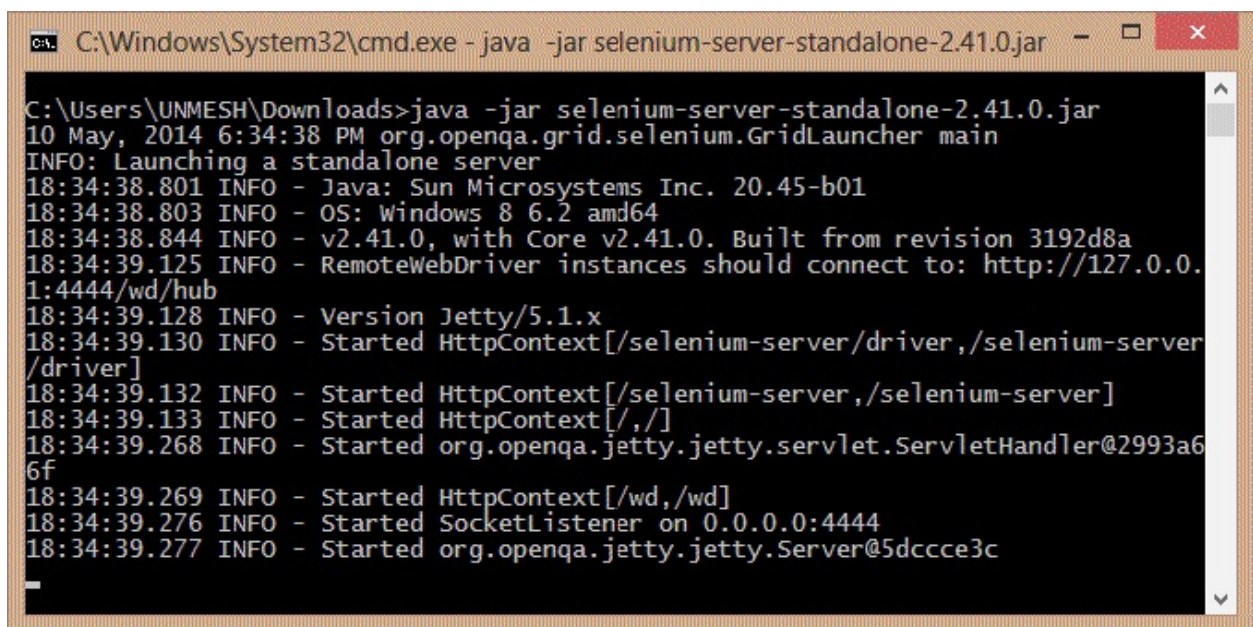
Selenium Standalone Server是用Java语言开发的，自我独立。在运行的时候，机器上需要安装JRE(Java Runtime Environment)。在运行Selenium Standalone Server之前，要确保远程机器上已经安装了JRE6或者更高的版本。

6.1.2 启动Selenium Standalone Server

Selenium Standalone Server能以不同的模式或角色启动，在本章节我们采用的是Standalone模式启动。可以通过在远程机器上保存有Selenium Standalone Server的JAR包文件的目录下启动命令行，使用以下命令启动Selenium Server，这是在Windows 8中启动Selenium Standalone Server的命令。

```
java -jar selenium-server-standalone-2.41.0.jar
```

Selenium Server启动后，默认监听端口号是4444) (http://<remote-machine-ip>:444)。在启动服务时，可以通过命令行更改端口号。下图是Selenium Server启动时的命令行输出。



```
C:\Windows\System32\cmd.exe - java -jar selenium-server-standalone-2.41.0.jar -
C:\Users\UNMESH\Downloads>java -jar selenium-server-standalone-2.41.0.jar
10 May, 2014 6:34:38 PM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a standalone server
18:34:38.801 INFO - Java: Sun Microsystems Inc. 20.45-b01
18:34:38.803 INFO - OS: Windows 8 6.2 amd64
18:34:38.844 INFO - v2.41.0, with Core v2.41.0. Built from revision 3192d8a
18:34:39.125 INFO - RemoteWebDriver instances should connect to: http://127.0.0.
1:4444/wd/hub
18:34:39.128 INFO - Version Jetty/5.1.x
18:34:39.130 INFO - Started HttpContext[/selenium-server/driver,/selenium-server
/driver]
18:34:39.132 INFO - Started HttpContext[/selenium-server,/selenium-server]
18:34:39.133 INFO - Started HttpContext[/,/]
18:34:39.268 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@2993a6
6f
18:34:39.269 INFO - Started HttpContext[/wd,/wd]
18:34:39.276 INFO - Started SocketListener on 0.0.0.0:4444
18:34:39.277 INFO - Started org.openqa.jetty.jetty.Server@5dccce3c
```

Selenium Server在远程机器上是以HTTP Server形式启动的，我们可以通过浏览器启动和查看该服务。在浏览器上输入http://<remote-machine-ip>:4444/wd/hub/static/resource/hub.html，就可以看到如下图所示的服务启动后的界面。



现在我们已经启动并运行Selenium Server，可以开始创建和执行测试了。

6.2 在Selenium Standalone Server上执行测试

要在Selenium Server上执行测试，我们需要使用RemoteWebDriver。这个Selenium Python中的Remote类以客户端的身份与Selenium Server进行交互，从而实现在远程机器上运行测试。我们使用这个类来指示Selenium Server做出相应的操作，以在远程机器上运行测试，以及在指定的浏览器上运行测试命令。

除了Remote类之外，我们需要设置desired_capabilities，即对浏览器和操作系统的配置，以及为了在Selenium Standalone Server上运行测试时要进行的一些其他配置。在此示例中，我们将指定运行测试的平台和浏览器名称，desired_capabilities配置如下。

```
desired_caps = {}
desired_caps['platform'] = 'WINDOWS'
desired_caps['browserName'] = 'firefox'
```

接下来，将创建一个Remote类的实例并传递desired_capabilities。当脚本执行时，该类将连接并请求Selenium Server启动Windows平台上的Firefox浏览器执行测试。

```
self.driver = webdriver.Remote('http://192.168.1.103:4444/wd/hub', desired_caps)
```

下面我们使用Remote类代替Firefox driver实现一个之前创建的搜索测试。

```
import unittest
from selenium import webdriver

class SearchProducts(unittest.TestCase):
    def setUp(self):
        desired_caps = {}
        desired_caps['platform'] = 'WINDOWS'
        desired_caps['browserName'] = 'firefox'

        self.driver = \
            webdriver.Remote('http://192.168.1.102:4444/wd/hub', desired_caps)
```

```

        self.driver.get('http://demo.magentocommerce.com
/')
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

    def testSearchByCategory(self):
        # get the search textbox
        self.search_field = self.driver.find_element_by_
name('q')
        self.search_field.clear()

        # enter search keyword and submit
        self.search_field.send_keys('phones')
        self.search_field.submit()

        # get all the anchor elements which have product
names
        # displayed currently on result page using
        # find_elements_by_xpath method
        products = self.driver\
            .find_elements_by_xpath('//h2[@class=\'produ
ct-name\']/a')

        # check count of products shown in results
        self.assertEqual(2, len(products))

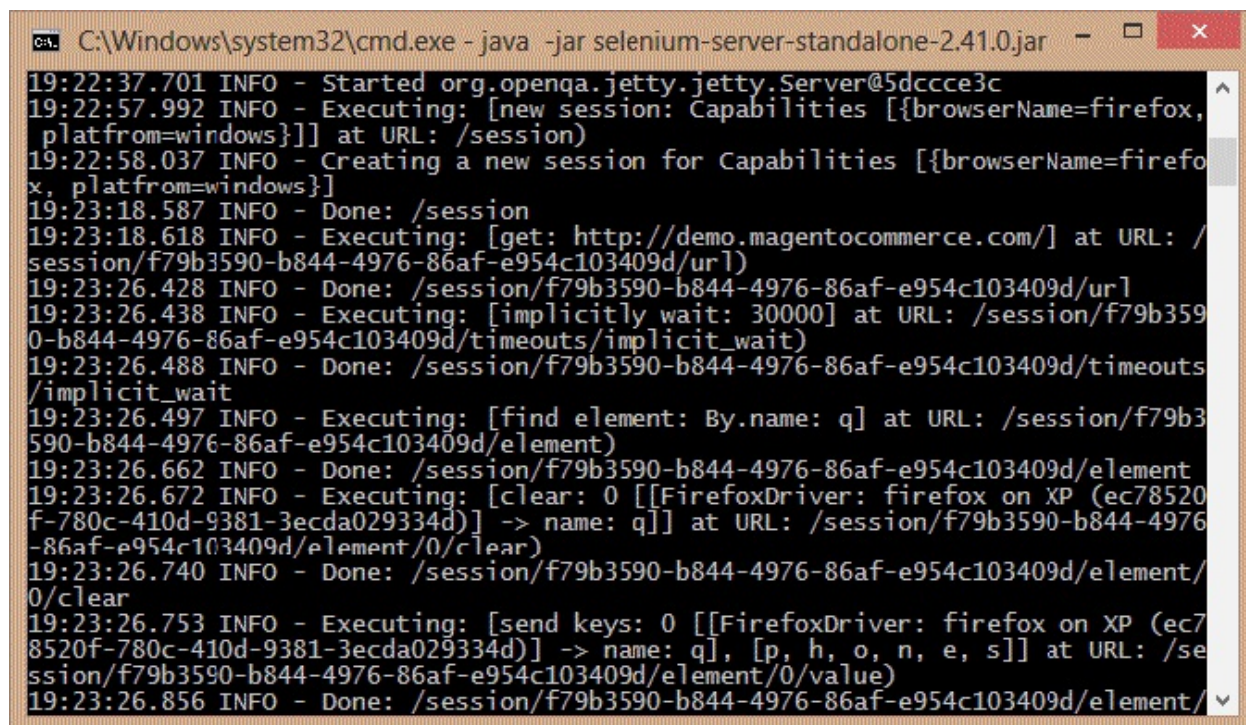
    def tearDown(self):
        # close the browser window
        self.driver.quit()

if __name__ == '__main__':
    unittest.main()

```

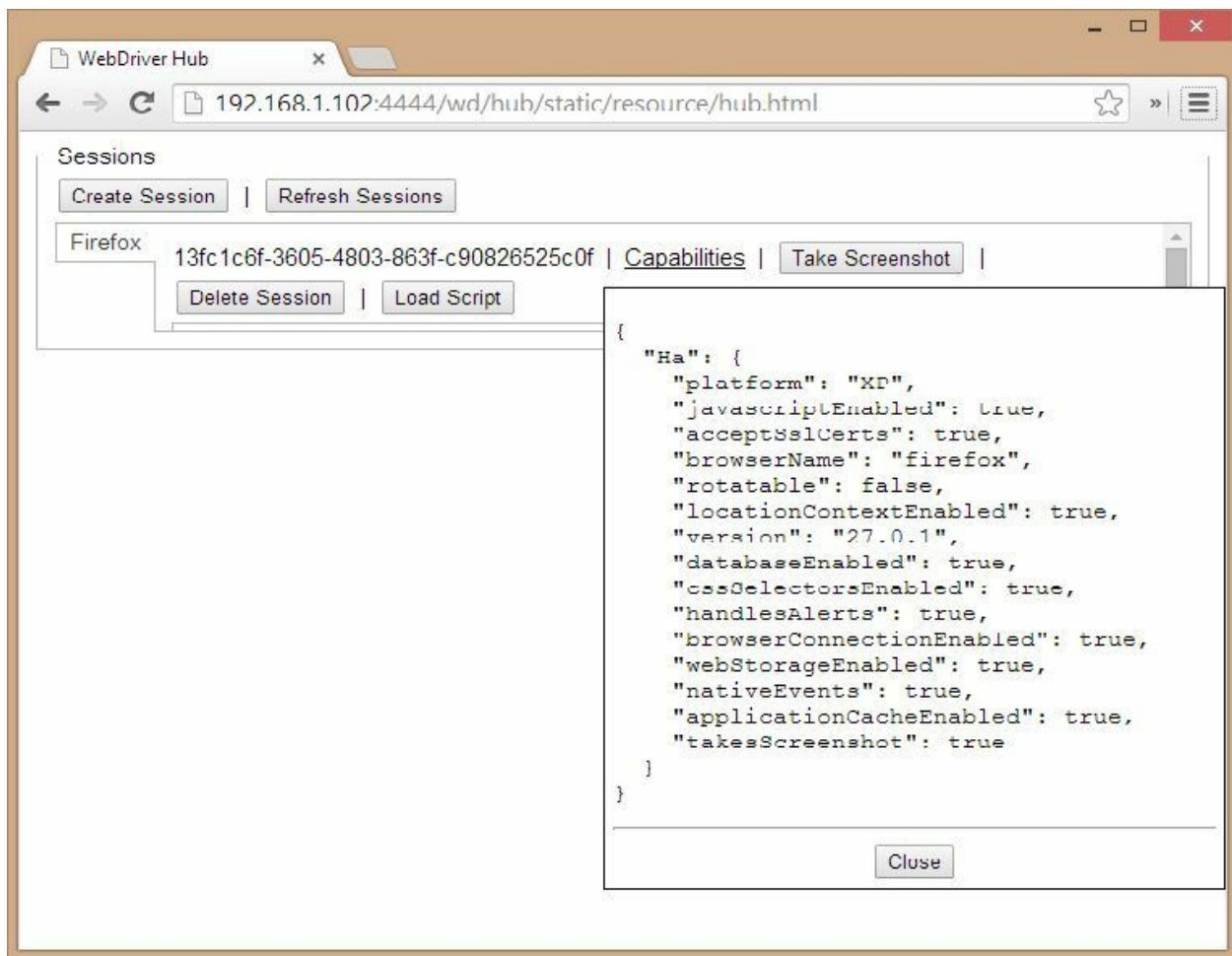
当执行此测试时，我们可以观察Selenium

Server控制台的输出。它可以实时显示测试脚本与Selenium Server之间是如何进行交互的，以及已执行的命令和返回状态。下图是测试执行时控制台的信息。

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe' and the command 'java -jar selenium-server-standalone-2.41.0.jar'. The window contains a series of log messages from the Selenium server, including session creation, URL execution, and element interactions. The logs are timestamped and include details about capabilities, session IDs, and specific commands like 'get', 'implicitly wait', 'find element', 'clear', and 'send keys'.

```
C:\Windows\system32\cmd.exe - java -jar selenium-server-standalone-2.41.0.jar
19:22:37.701 INFO - Started org.openqa.jetty.jetty.Server@5dccc3c
19:22:57.992 INFO - Executing: [new session: Capabilities [{browserName=firefox,
platform=windows}]] at URL: /session)
19:22:58.037 INFO - Creating a new session for Capabilities [{browserName=firefox,
platform=windows}]
19:23:18.587 INFO - Done: /session
19:23:18.618 INFO - Executing: [get: http://demo.magentocommerce.com/] at URL: /
session/f79b3590-b844-4976-86af-e954c103409d/url)
19:23:26.428 INFO - Done: /session/f79b3590-b844-4976-86af-e954c103409d/url
19:23:26.438 INFO - Executing: [implicitly wait: 30000] at URL: /session/f79b359
0-b844-4976-86af-e954c103409d/timeouts/implicit_wait)
19:23:26.488 INFO - Done: /session/f79b3590-b844-4976-86af-e954c103409d/timeouts
/implicit_wait
19:23:26.497 INFO - Executing: [find element: By.name: q] at URL: /session/f79b3
590-b844-4976-86af-e954c103409d/element)
19:23:26.662 INFO - Done: /session/f79b3590-b844-4976-86af-e954c103409d/element
19:23:26.672 INFO - Executing: [clear: 0 [[FirefoxDriver: firefox on XP (ec78520
f-780c-410d-9381-3ecda029334d)] -> name: q]] at URL: /session/f79b3590-b844-4976
-86af-e954c103409d/element/0/clear)
19:23:26.740 INFO - Done: /session/f79b3590-b844-4976-86af-e954c103409d/element/
0/clear
19:23:26.753 INFO - Executing: [send keys: 0 [[FirefoxDriver: firefox on XP (ec7
8520f-780c-410d-9381-3ecda029334d)] -> name: q], [p, h, o, n, e, s]] at URL: /se
ssion/f79b3590-b844-4976-86af-e954c103409d/element/0/value)
19:23:26.856 INFO - Done: /session/f79b3590-b844-4976-86af-e954c103409d/element/
```

在浏览器的[http:// <remote-machine-ip>:4444/wd/hub/static/resource/hub.html](http://<remote-machine-ip>:4444/wd/hub/static/resource/hub.html)页面可以发现，一个新的会话已创建。当鼠标停留在Capabilities链接上时，将显示用于该测试的Capabilities详细信息，如下图所示。



6.2.1 配置IE支持

Selenium Server与Firefox绑定，默认支持Firefox，但是要想在Internet Explorer（IE）上执行测试，就需要在启动Selenium Server时指定IE driver可执行文件的路径。下面是在命令行中通过配置webdriver.ie.driver选项来指定IE driver可执行文件路

径的命令。

```
java -Dwebdriver.ie.driver="C:\SeDrivers\IEDriverServer.exe" -jar selenium-server-standalone-2.41.0.jar
```

通过指定IE driver路径启动Selenium Server后，就可以在远程机器的IE浏览器上执行测试了。

6.2.2 配置Chrome支持

与IE driver类似，要想在Chrome上执行测试，就需要指定Chrome driver可执行文件。下面是通过配置webdriver.chrome.driver选项指定Chrome driver可执行文件路径的命令。

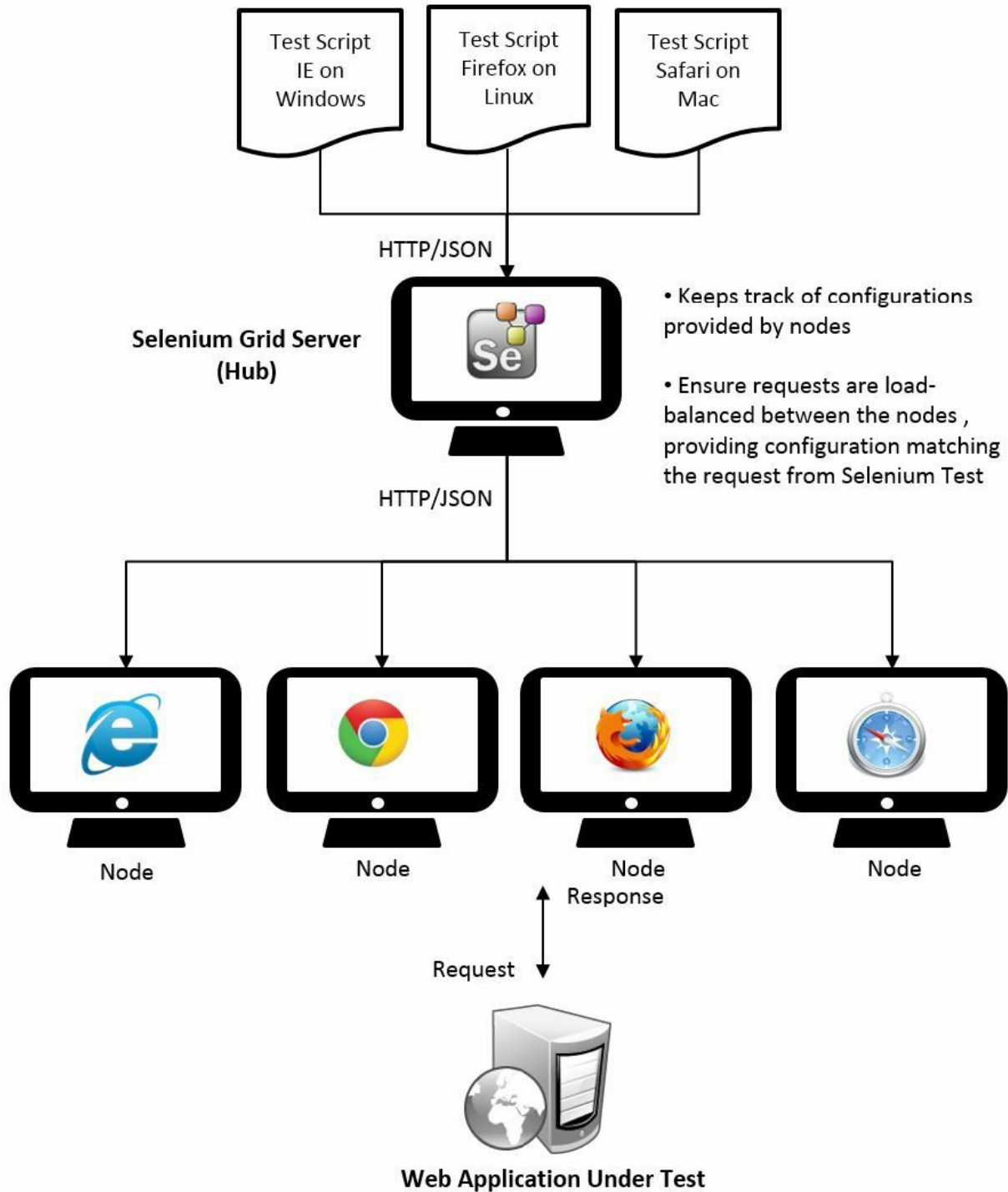
```
java -Dwebdriver.ie.driver="C:\SeDrivers\IEDriverServer.exe" -Dwebdriver.chrome.driver="C:\SeDrivers\chromedriver.exe" -jar selenium-server-standalone-2.41.0.jar
```

此时，Selenium Server已同时支持在远程机器的Internet Explorer和Chrome浏览器上执行测试。

6.3 Selenium Grid

Selenium Grid可以将测试分布在若干个物理或虚拟机器上，从而实现分布方式或并行方式执行测试。这样可以有效减少执行测试所需周期，同时实现跨浏览器测试来获得更快、更准确的结果反馈。我们可以使用云端现有的虚拟机建立Grid。

Selenium Grid能够在若干个节点或客户端上并行执行多个测试，这些节点或客户端都可以是不同浏览器和操作系统，从而支持混合的测试环境。Grid使所有节点如下图展示的那样，在底层独立且透明地实现分布测试。



6.3.1 启动hub

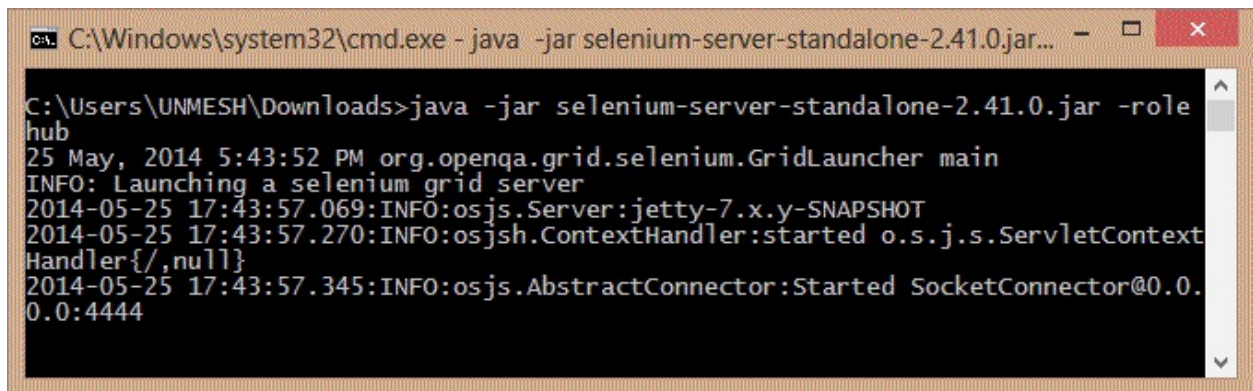
在分布式测试中，启动Selenium Server作为一个hub，hub提供所有可用的配置或属性信息给要执行的测试，然后slave机器（也称为节点）将连接到这个hub。测试脚本运用JSON Wire Protocol，并通过Remote类与hub交互来执行Selenium命令。更多关于JSON Wire Protocol的信息可以访问<https://code.google.com/p/selenium/wiki/JsonWireProtocol>

hub作为中心节点，接收测试命令并将它们分发给适当节点或符合匹配要求的节点。下面我们将Selenium Server配置成Grid，然后配置一些不同浏览器和操作系统组合的节点。

用前面章节中学到的命令并添加一些参数，就可以将Selenium Standalone Server作为hub（也称为Grid Server）启动。首先打开一个新的命令行/终端窗口，然后定位到Selenium Server JAR所在的位置。输入以下命令以hub形式启动Server。

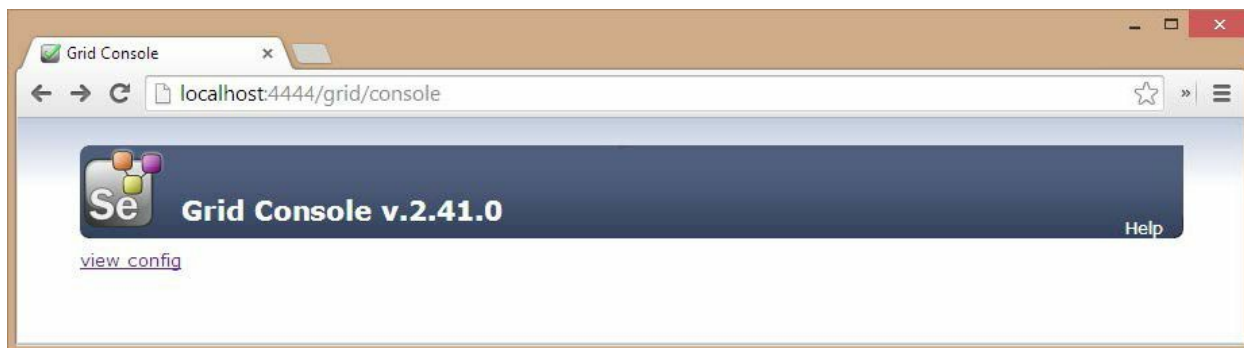

```
java -jar selenium-server-standalone-2.25.0.jar -port 4444 -role hub
```

注意：要将Server配置成hub或者Grid Server，那么在启动时就需要指定`-role`参数，值为“hub”。本例中，我们是在Windows系统中启动Server。下图是启动过程中控制台打印的信息。



```
C:\Windows\system32\cmd.exe - java -jar selenium-server-standalone-2.41.0.jar... - □ ×  
C:\Users\UNMESH\Downloads>java -jar selenium-server-standalone-2.41.0.jar -role  
hub  
25 May, 2014 5:43:52 PM org.openqa.grid.selenium.GridLauncher main  
INFO: Launching a selenium grid server  
2014-05-25 17:43:57.069:INFO:osjs.Server:jetty-7.x.y-SNAPSHOT  
2014-05-25 17:43:57.270:INFO:osjs.ContextHandler:started o.s.j.s.ServletContext  
Handler{/,null}  
2014-05-25 17:43:57.345:INFO:osjs.AbstractConnector:Started SocketConnector@0.0.  
0.0:4444
```

当我们以hub形式启动Server之后，它就是一个Grid Server。我们可以通过浏览器查看Grid控制台的信息，如下图所示。



6.3.2 添加节点

现在我们将Selenium Server作为Grid Server启动，接下来是在Server上添加并配置节点。

6.3.2.1 添加IE节点

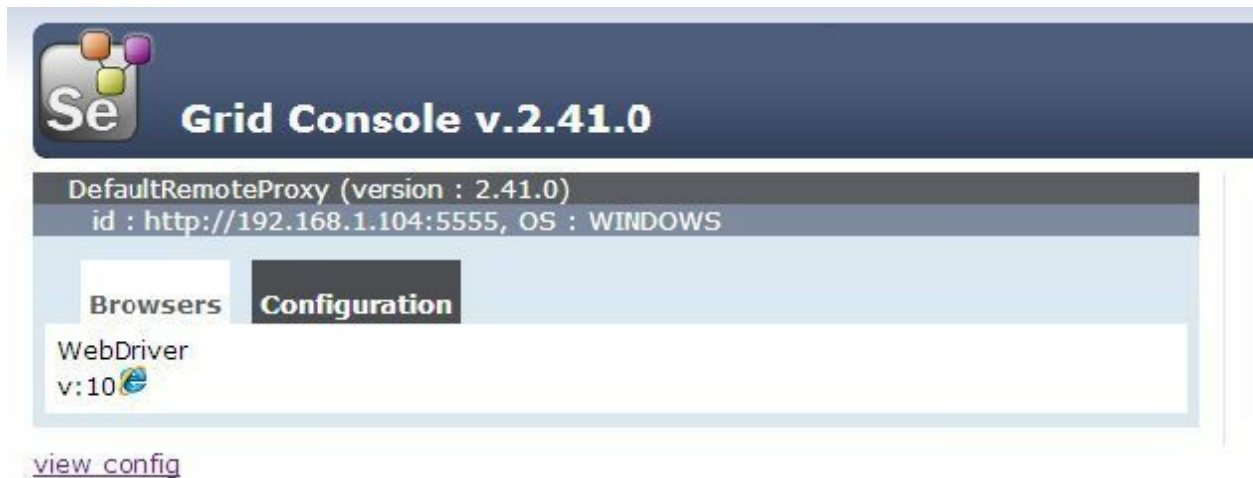
先从安装有Internet Explorer的Windows节点开始添加。打开新的命令行或终端窗口并定位到Selenium Server JAR包文件所在的目录。输入以下命令启动节点并将它添加至Grid。

```
java -Dwebdriver.ie.driver="C:\SeDrivers\IEDriverServer.exe" -jar selenium-server-standalone-2.41.0.jar -role webdriver -browserName=internet explorer,version=10,maxinstance=1,platform=WINDOWS -hubHost 192.168.1.103 -port 5555
```


要将节点添加到Grid，我们需要使用-role参数并传递值“webdriver”，还需要使用-browser参数配置浏览器信息。在这个例子中，通过browserName指定浏览器为Internet Explorer，版本为10，maxinstance为1，平台为Windows。其中maxinstance值是告诉Grid可以支持多少并发浏览器实例。

要将节点连接到hub或Grid Server，我们还需要指定-hubHost参数与Grid Server的主机名或IP地址。最后，指定节点就可以连接到hub上对应的端口。

当我们执行上述命令后，节点将被启动，Grid控制台上会出现如下图所示的配置。



除了上面的方法外，也可以通过JSON格式的配置文件来添加节点。JSON配置文件代码如下。

```
{
  "class": "org.openqa.grid.common.RegistrationRequest"
,
  "capabilities": [
    {
      "seleniumProtocol": "WebDriver",
      "browserName": "internet explorer",
      "version": "10",
      "maxInstances": 1,
      "platform" : "WINDOWS"
    }
  ],
  "configuration": {
    "port": 5555,
    "register": true,
    "host": "192.168.1.103",
    "proxy": "org.openqa.grid.selenium.proxy. DefaultRemoteProxy",
    "maxSession": 2,
  }
}
```

```
"hubHost": "192.168.1.100",  
"role": "webdriver",  
"registerCycle": 5000,  
"hub": "http://192.168.1.100:4444/grid/register",  
"hubPort": 4444,  
"remoteHost": "http://192.168.1.102:5555"  
}  
}
```

我们可以在命令行参数中传递JSON配置文件“selenium-node-win-ie10.cfg.json”，下面就是通过JSON配置文件方式启动Server的命令。

```
java -Dwebdriver.ie.driver="C:\\SeDrivers\\IEDriverServer  
.exe"-jar  
selenium-server-standalone-2.41.0.jar -role webdriver -  
nodeConfig  
selenium-node-win-ie10.cfg.json
```

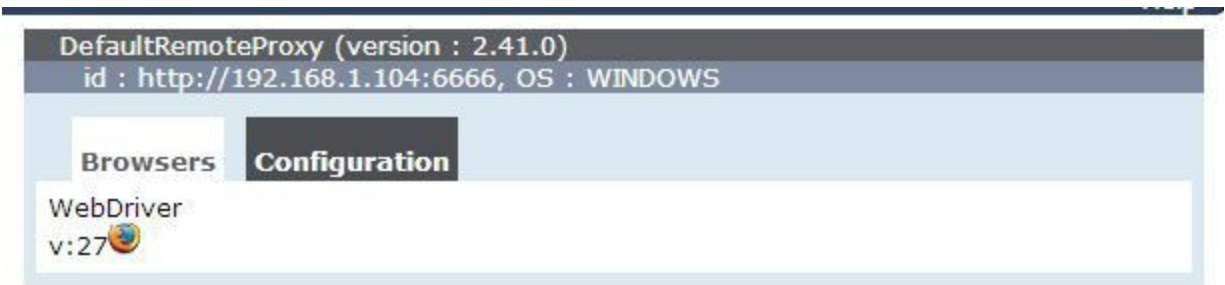
6.3.2.2 添加Firefox节点

现在开始添加Firefox节点。打开新的命令行或终端窗口，定位到Selenium Server JAR包文件所在目录下，输入以下命令启动节点并添加至Grid。

```
java -jar selenium-server-standalone-2.41.0.jar -role w  
ebdriver -browser
```

```
"browserName=firefox,version=27,maxinstance=2,platform=WINDOWS" -hubHost localhost -port 6666
```

在此，我们设置maxinstance值为2，也就是告诉Grid可以同时支持两个Firefox实例。Firefox节点启动后，Grid控制台就会出现如下图所示的配置。

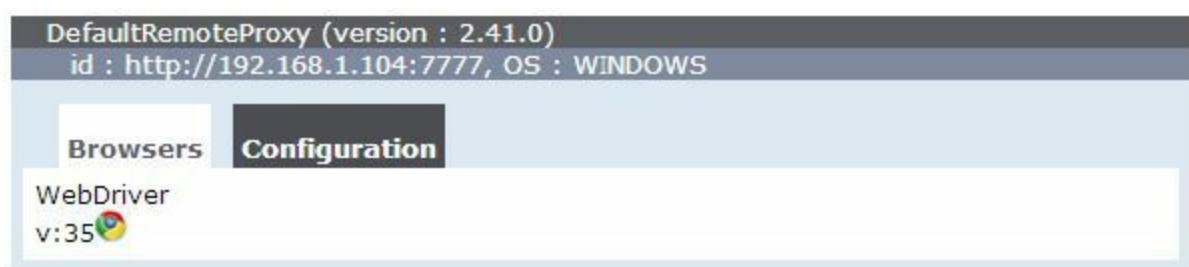


6.3.2.3 添加Chrome节点

接下来添加Chrome节点。打开新的命令行或终端窗口，定位到Selenium Server JAR包文件所在目录下，输入以下命令启动节点并添加至Grid。

```
java -Dwebdriver.chrome.driver="C:\SeDrivers\chromedriver.exe" -jar selenium-server-standalone-2.41.0.jar -role webdriver -browser "browserName=chrome,version=35,maxinstance=2,platform=WINDOWS" -hubHost localhost -port 7777
```

下图是Chrome节点启动后Grid控制台出现的配置信息。



6.4 Mac OS X的Safari节点

我们已经从Windows系统添加了IE、Firefox和Chrome节点，现在我们要从Mac系统添加一个Safari节点。打开一个新的终端窗口，定位到Selenium Server JAR包文件所在目录下，输入以下命令启动节点并添加至Grid。

```
java -jar selenium-server-standalone-2.41.0.jar -role w  
ebdriver -browser  
"browserName=safari,version=7,maxinstance=1,platform=MA  
C" -hubHost 192.168.1.104 -port 8888
```

下图是该节点启动后Grid控制台上出现的所有配置信息。



现在，我们配置好了Selenium Grid，让我们尝试在这个Grid上执行测试吧。

6.5 在Grid上执行测试

在配置有不同浏览器和操作系统组合的Grid上执行测试之前，需要对我们之前创建的测试进行一些调整。之前设置desired_capabilities的时候，我们硬编码指定了浏览器和平台名称。如果这些值被硬编码了，那么我们就需要为每个组合都单独准备一个测试脚本。为了避免这一点而使用同一个测试脚本来测试所有的组合，我们需要参数化浏览器和平台名称这两个值，然后传递到desired_capabilities类中，如以下步骤中所示。

(1) 从命令行中将浏览器和平台的值传递给测试脚本。例如，如果要在Windows和Chrome组合上执行测试，可以通过以下方式在命令行中运行脚本。

```
python grid_test.py WINDOWS chrome
```


(2) 如果要在Mac和Safari组合上执行测试，使用以下命令运行脚本。

```
python grid_test.py MAC safari
```

(3) 要实现这一点，需要在以下的测试类中添加PLATFORM和BROWSER两个全局属性，并分别设置一个默认值，以防从命令行运行脚本时没有提供值。

```
class SearchProducts(unittest.TestCase):  
    PLATFORM = 'WINDOWS'  
    BROWSER = 'firefox'
```

(4) 接下来，在setUp()方法中参数化这些desired_capabilities，如下面的代码所示。

```
desired_caps = {}  
desired_caps['platform'] = self.PLATFORM  
desired_caps['browserName'] = self.BROWSER
```

(5) 最后，读取命令行参数值并传递给脚本，从而为PLATFORM和BROWSER属性赋值。

```
if __name__ == '__main__':
    if len(sys.argv) > 1:
        SearchProducts.BROWSER = sys.argv.pop()
        SearchProducts.PLATFORM = sys.argv.pop()
    unittest.main()
```

(6) 就是这样。现在我们的测试已经准备好处理任何给定的环境组合了。以下是完整代码。

```
import sys
import unittest
from selenium import webdriver

class SearchProducts(unittest.TestCase):
    PLATFORM = 'WINDOWS'
    BROWSER = 'firefox'

    def setUp(self):
        desired_caps = {}
        desired_caps['platform'] = self.PLATFORM
        desired_caps['browserName'] = self.BROWSER

        self.driver = \
            webdriver.Remote('http://192.168.1.104:4444/wd/hub', desired_caps)
        self.driver.get('http://demo.magentocommerce.com/')

        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

    def testSearchByCategory(self):
        # get the search textbox
```

```

        self.search_field = self.driver.find_element_by_
name('q')
        self.search_field.clear()

        # enter search keyword and submit
        self.search_field.send_keys('phones')
        self.search_field.submit()

        # get all the anchor elements which have product
names
        # displayed currently on result page using
        # find_elements_by_xpath method
        products = self.driver.\
            find_elements_by_xpath('//h2[@class=\'produc
t-name\']/a')

        # check count of products shown in results
        self.assertEqual(2, len(products))

    def tearDown(self):
        # close the browser window
        self.driver.quit()

if __name__ == '__main__':
    if len(sys.argv) > 1:
        SearchProducts.BROWSER = sys.argv.pop()
        SearchProducts.PLATFORM = sys.argv.pop()
    unittest.main(verbosity=2)

```

(7) 打开新的命令行或终端窗口，定位至脚本所在位置的目录，输入以下命令执行测试。你将看到Grid会自动连接到与给定平台和浏览器匹配的

节点并在该节点上执行测试。

```
python grid_test.py MAC safari
```

6.6 在云端执行测试

为了实现跨浏览器测试，我们在之前的步骤中搭建了本地Grid。搭建本地Grid需要给物理或虚拟机配置不同浏览器和操作系统。可是获取这些硬件和软件设备是需要很大的成本和努力的，而且你还需要在这些设备的更新和补丁等方面投入大量的精力，这不是每个人或团队都能负担得起的。

但现在你可以轻松地将虚拟测试lab外包给第三方云测试提供商，而不必花费精力投资和搭建跨浏览器测试lab。Sauce Labs和BrowserStack是领先的基于云端的跨浏览器测试云提供商。这两个都支持超过400种不同的浏览器和操作系统配置，包括移动和平板设备，并支持在他们的云端环境中运行Selenium WebDriver测试。

在本节中，我们将在Sauce Labs中安装并执行

测试。用BrowserStack执行测试的步骤也是类似的。

下面我们使用Sauce Labs搭建并执行测试，步骤如下。

（1）首先需要有一个免费的Sauce Labs账号。可以从Sauce Labs官网（<https://saucelabs.com/>）注册免费账号，获取用户名和访问密钥。Sauce Labs在云端环境中提供了执行测试所需的所有硬件和软件等基础配置。

（2）登录后从Sauce Labs主页获取访问密钥，如下图所示。



（3）修改之前Grid中运行时所创建的测试，

并添加步骤使其可以运行在Sauce Labs上。

(4) 在测试脚本中添加Sauce用户名和访问密钥，更改Grid地址为Sauce Grid地址并传递该用户名和访问密钥，如下面代码所示。

```
import sys
import unittest
from selenium import webdriver

class SearchProducts(unittest.TestCase):
    PLATFORM = 'WINDOWS'
    BROWSER = 'phantomjs'
    SAUCE_USERNAME = 'upgundecha'
    SUACE_KEY = 'c6e7132c-ae27-4217-b6fa-3cf7df0a7281'

    def setUp(self):
        desired_caps = {}
        desired_caps['platform'] = self.PLATFORM
        desired_caps['browserName'] = self.BROWSER

        sauce_string = self.SAUCE_USERNAME + ':' + self.
SUACE_KEY

        self.driver = \
            webdriver.Remote('http://' + sauce_string +
                '@ondemand.saucelabs.com:80/wd/hub', desir
ed_caps)
        self.driver.get('http://demo.magentocommerce.com
/')
        self.driver.implicitly_wait(30)
```

```

        self.driver.maximize_window()

    def testSearchByCategory(self):
        # get the search textbox
        self.search_field = self.driver.find_element_by_
name('q')
        self.search_field.clear()

        # enter search keyword and submit
        self.search_field.send_keys('phones')
        self.search_field.submit()

        # get all the anchor elements which have product
names
        # displayed currently on result page using
        # find_elements_by_xpath method
        products = self.driver.\
            find_elements_by_xpath('//h2[@class=\'produc
t-name\']/a')

        # check count of products shown in results
        self.assertEqual(2, len(products))

    def tearDown(self):
        # close the browser window
        self.driver.quit()

if __name__ == '__main__':
    if len(sys.argv) > 1:
        SearchProducts.BROWSER = sys.argv.pop()
        SearchProducts.PLATFORM = sys.argv.pop()
    unittest.main(verbosity=2)

```

(5) 打开命令行或终端窗口，进入脚本所在

目录，输入以下命令执行测试。

```
python sauce_test.py "OS X 10.9" "Safari"
```



可以在<https://saucelabs.com/platforms>获取 Sauce Labs 支持的平台列表。

测试执行时，它将连接到Sauce Labs的Grid Server并请求所需的操作系统和浏览器配置。Sauce 为我们的测试分配虚拟机并在给定的配置上运行。

（6）我们可以在Sauce Dashboard上监视执行状态，如下图所示。

Session	Environment	Tags	Build	Results	End ▾	Run Time
<input type="checkbox"/> unnamed job	🍏 10.9 🌐 7			Running View Fullscreen Spy		




我们还可以在Sauce会话部分进一步了解测试执行过程中究竟发生了什么。Sauce会话页面显示

了测试执行的细节，包括Selenium命令、截图、Selenium日志以及执行过程的截屏，如下图所示。

unnamed job
complete

by upgundecha
Start Time: 10-12-2014 23:15:01 | Duration: 24s

10.9 7 Not on Sauce Connect

Commands	Screenshot	Selenium Log
<div>POST /session</div> <div>desiredCapabilities:{"browserName":"safari","safari.options":{"port":1034}}</div> <div>=> {"takesScreenshot":true,"cssSelectorsEnabled":true,"ja...</div> <div>1s (+3.78s)</div>		<div>SCR</div> <div></div>
<div>POST url</div> <div>url: "http://demo.magentocommerce.com/"</div> <div>=> ""</div> <div>7s (+2.03s)</div>		
<div>POST timeouts/implicit_wait</div> <div>mc: 30000</div> <div>10s (+0.02s)</div>		



也可以用Sauce Connect在内部服务器上更安全地测试你的应用程序。Sauce Connect会在你的机器和Sauce云中创建一个安全的通道。

6.7 章节回顾

在本章中，我们学习了如何使用Selenium Standalone Server在远程机器上执行测试。Selenium Standalone Server能够在远程机器上对应用程序进行任意浏览器和操作系统组合的跨浏览器测试。这不但增加了测试覆盖率，而且能确保应用程序在期望的组合上执行。

然后，我们搭建了Selenium Grid，并在分布式架构中执行测试。Selenium Grid可以在对多个机器上透明地执行，从而降低跨浏览器测试的复杂性，也减少了测试执行时间。

我们还考虑了基于云端的跨浏览器测试提供商，在Sauce Labs上执行测试。Sauce Labs提供了执行测试所需的所有基础配置，支持上百种不同的组合，成本更低。

在下一章中，我们将学习如何使用Appium和Selenium WebDriver测试移动端应用程序，在此过程中会用到一些本章中学习到的概念。Appium支持在iOS和Android系统上测试原生、混合以及移动Web应用程序。我们将展示用Appium针对移动端测试的示例。

第7章 移动端测试

随着全世界移动端用户数量的不断增加，智能手机和平板电脑的使用者的数量已经显著增加。智能设备正在逐渐取代台式机和笔记本电脑，移动端应用程序已经渗透到消费者与企业市场。无论是小企业还是大企业，在使用移动端作为连接用户的渠道方面都潜力巨大。大家都努力建设显示友好的移动端网站或者App（应用程序）来服务客户和内部员工。测试这些应用能否在市场上流行的各种移动端设备上正常运行变得至关重要。本章将讲解如何使用WebDriver和Appium来测试移动端应用程序。

本章包含以下主题：

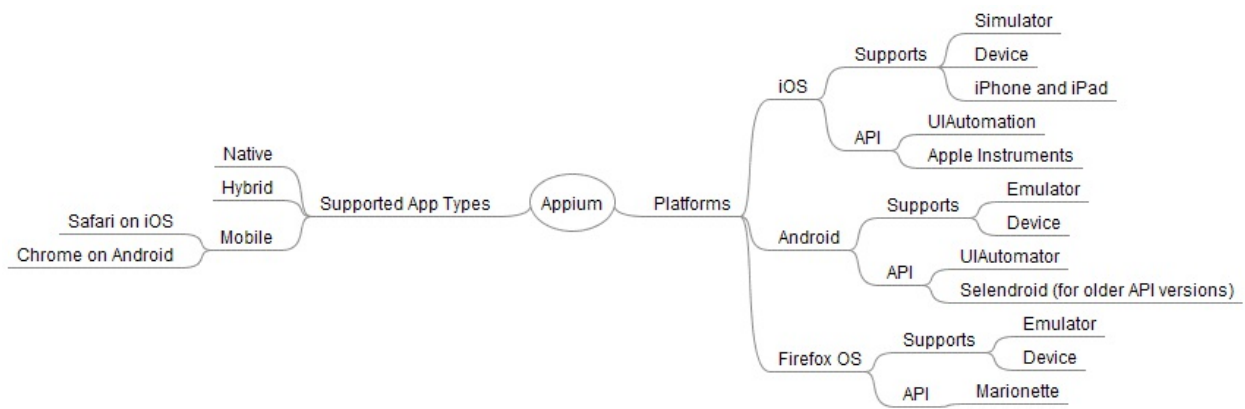
- 如何使用Appium测试移动端应用；
- Appium的安装和配置；

- 在iPhone模拟器上创建并运行iOS测试;
- 在真机上创建并运行Android测试。

7.1 认识Appium

Appium是一个开源的自动化测试框架，可以用来测试基于iOS、Android和Firefox OS 平台的原生与混合的应用。该框架使用Selenium WebDriver，在执行测试时用于和Selenium Server 通信的是JSON Wire Protocol。在Selenium 2中，Appium将取代 iPhoneDriver 和 AndroidDriver API，并用于测试移动互联网应用程序。

Appium允许我们使用，甚至扩展现有的Selenium WebDriver 框架来构建测试脚本。由于Appium是通过Selenium WebDriver 来驱动测试脚本的，因此只要有对应的Selenium client library存在，就可以使用相应的语言来创建测试脚本。下图是Appium对不同平台和应用类型的支持情况的覆盖地图。



7.1.1 Appium支持的应用类型

Appium 支持以下应用类型的测试。

- 原生应用。原生应用是指适用于特定平台的，即使用该平台所支持的语言和框架来构建的。例如，iPhone和iPad上面的应用都是使用Objective-C和iOS SDK 来开发的；同样，Android 应用是使用Java和Android SDK来开发的。在程序运行的时候，原生应用会更加流畅和稳定。它们是使用原生框架来构建用户交互界面。
- 移动端**Web**应用。移动端Web应用是服务端应用，是使用PHP、Java或者ASP.NET这样的服务

端技术来构建的，并且使用jQuery Mobile、Sencha Touch 等一些技术渲染用户页面以模拟本地UI。

- 混合应用。类似于原生应用程序，混合应用是运行在移动设备上并且通过一些互联网技术（HTML5、CSS和JavaScript）来实现的。混合应用使用移动设备的浏览器引擎来渲染HTML页面，并且通过使用WebView在本地容器中处理JavaScript脚本。这种处理方式可以使混合应用具备访问一些移动Web应用不能访问的设备（比如相机、加速计、传感器和本地存储器）的能力。

7.1.2 Appium环境准备

在开始学习更多关于Appium的知识之前，首先需要了解一些基于iOS和Android平台的工具。



Appium是基于Node.js实现的，在Mac OS X和Windows平台上都有对应的Standalone GUI的Node.js 包。我们可以使用Mac OS X平台上内置在Node.js框架内的Appium Standalone GUI。

7.1.2.1 安装Xcode

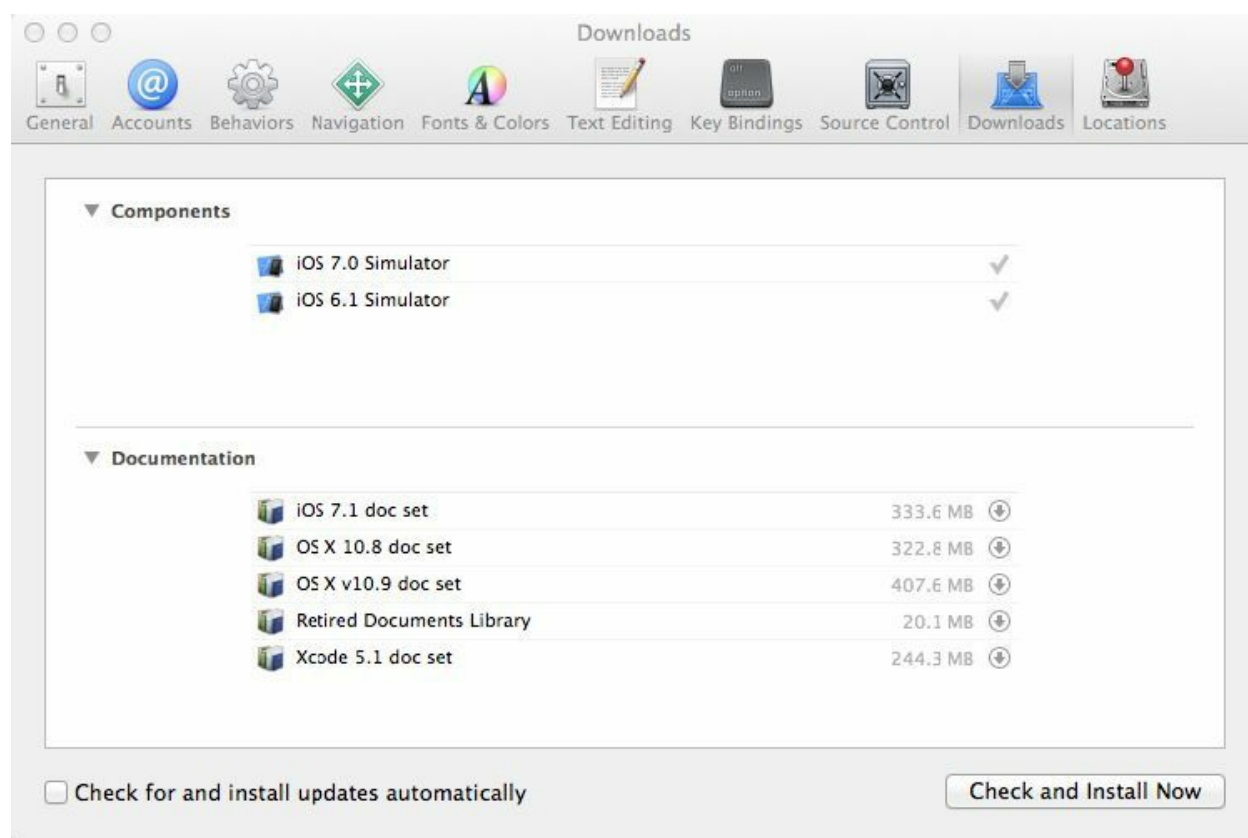
我们需要在Mac OS X系统上使用Xcode 4.6.3或者更高的版本，来测试iOS平台下的应用。

写本书时，使用的是Xcode 5.1版本。可以在App Store 或者苹果开发者网站下载：

<https://developer.apple.com/xcode/>。

安装完Xcode后，从应用程序菜单启动它，然后单击Preferences | Downloads。安装Command Line Tools和其他的iOS SDK，用来测试在不同版本iOS

平台下的应用程序，如下图所示。



为了在真机上运行测试脚本，需要首先安装好 Provision Profile，并且打开USB调试模式。

启动iPhone模拟器并验证其是否正常工作。可以通过菜单Xcode | Open Developer Tool | iOS Simulator 来启动模拟器。在模拟器中启动Safari浏览器，并且打开如下图所示Web应用的样例。

7.1.2.2 安装Android SDK

我们需要安装Android SDK来测试Android应用。从以下网址可以获取最新版本的Android SDK: <http://developer.android.com/sdk/>。安装完成以后, 确保ANDROID_HOME已经成功添加到环境变量的path中。


完整安装步骤参考:

<http://developer.android.com/sdk/installing/index.html?pkg=tools>。



有关最新的Appium安装要求和细节, 请访问相关网站了解。




demo.magentocommerce.com 



This is a demo store. Any orders placed through this store will not be honored or fulfilled.

MADISON ISLAND

English 



Menu



WING
MAN

*hit the runway in stylish
separates and cauals*

SHOP MEN



HOME & DECOR
FOR ALL YOUR SPACES

SHOP PRIVATE SALES



7.1.2.3 安装Appium Python client

在完成此书时，Appium Python client是完全符合Selenium 3.0规范草案的。这有助于更加方便地使用Python和Appium来编写移动端测试脚本。可以通过以下命令来安装Appium Python client。

```
pip install Appium-Python-Client
```



可以访问以下网址查看更多关于Appium Python client 安装包的信息：
<https://pypi.python.org/pypi/Appium-Python-Client>。

7.2 安装Appium

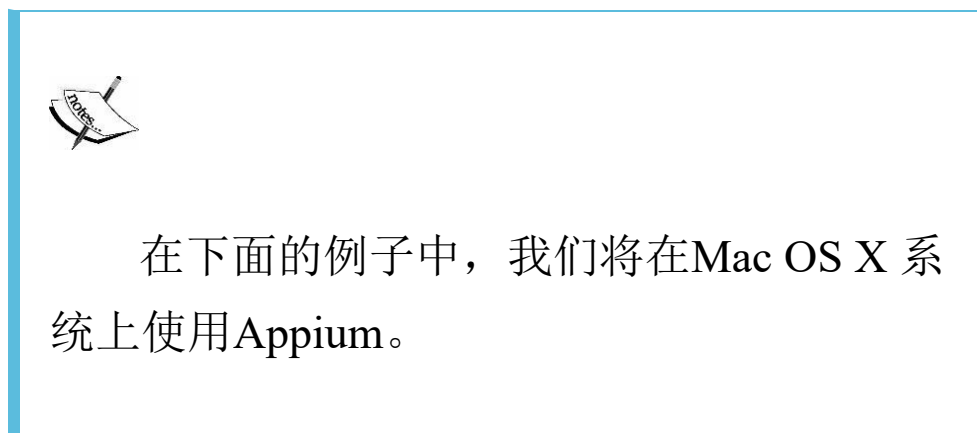
在使用Appium测试移动应用之前，我们首先需要下载和安装Appium。我们选择安装Appium GUI版。如果希望在iPhone或者iPad上运行iOS测试，那么就需要在装有Mac OS X系统的机器上安装Appium。如果是测试Android应用程序，需要在装有Windows或者Linux系统的机器上安装Appium。在Mac OS X系统上安装Appium是非常简单的。可以从以下网址下载最新版本的Appium安装文件：<http://appium.io/>。

具体安装步骤如下。

(1) 在网站首页单击**Download Appium**按钮，即可直接跳转到下载页面，如下图所示。

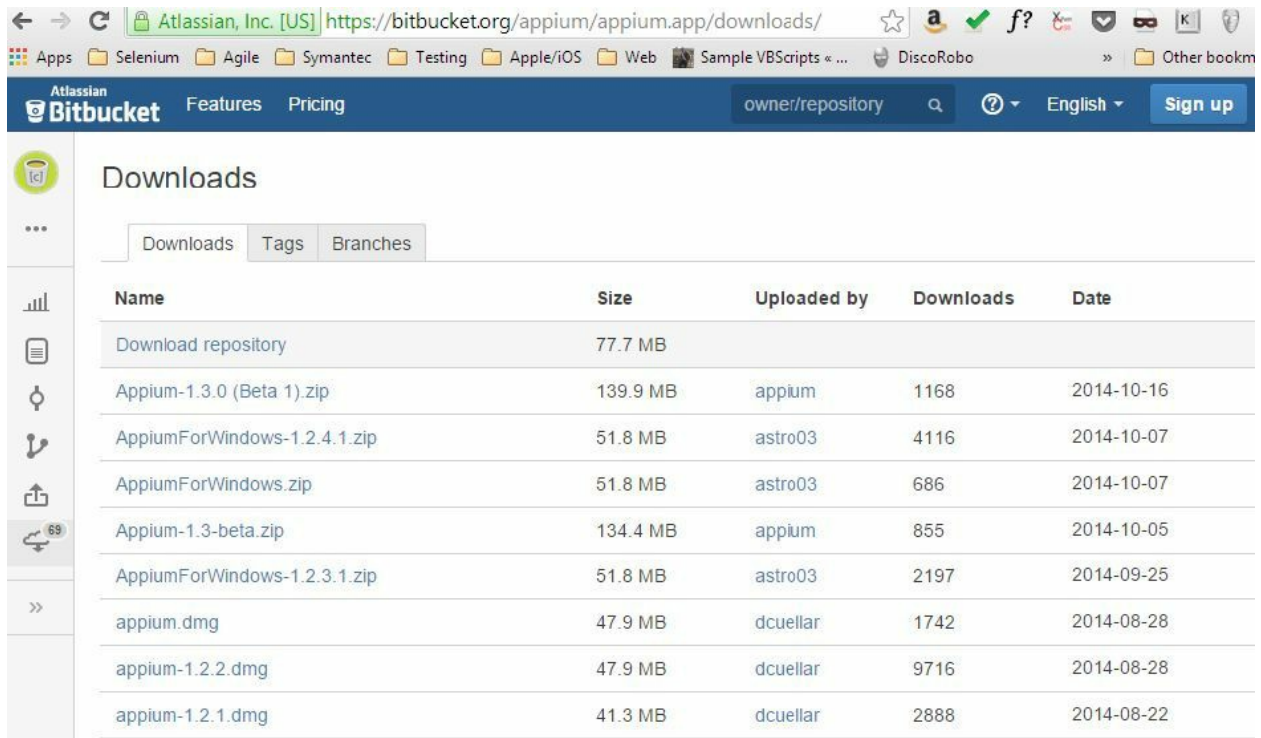


(2) 从列表中选择你正在使用的操作系统对应的安装版本，如下图所示。



(3) 在Mac系统中，可以通过运行安装程序来安装Appium，并且把Appium复制到Applications

目录下。



The screenshot shows the Bitbucket web interface for the 'appium' repository. The 'Downloads' tab is selected, displaying a table of download files. The table has columns for Name, Size, Uploaded by, Downloads, and Date. The files listed include various versions of Appium for Windows and macOS, as well as the repository itself.

Name	Size	Uploaded by	Downloads	Date
Download repository	77.7 MB			
Appium-1.3.0 (Beta 1).zip	139.9 MB	appium	1168	2014-10-16
AppiumForWindows-1.2.4.1.zip	51.8 MB	astro03	4116	2014-10-07
AppiumForWindows.zip	51.8 MB	astro03	686	2014-10-07
Appium-1.3-beta.zip	134.4 MB	appium	855	2014-10-05
AppiumForWindows-1.2.3.1.zip	51.8 MB	astro03	2197	2014-09-25
appium.dmg	47.9 MB	dcuellar	1742	2014-08-28
appium-1.2.2.dmg	47.9 MB	dcuellar	9716	2014-08-28
appium-1.2.1.dmg	41.3 MB	dcuellar	2888	2014-08-22

当第一次从Applications菜单启动Appium的时候，它将会要求授权来运行iOS模拟器。



默认情况下，Appium启动后的URL和端口是http://127.0.0.1:4723或localhost。该URL正是测试脚本发指令对应的地址。我们将基于iPhone的Safari浏览器来测试本书中的样例

程序的移动版本。

(4) 在如下图所示的Appium的主界面，单击Apple图标，打开iOS设置对话框。

(5) 在iOS设置对话框中，选中Force Device复选框，并且在下拉列表中选择iPhone型号。另外，选中Use Mobile Safari 复选框，如下图所示。

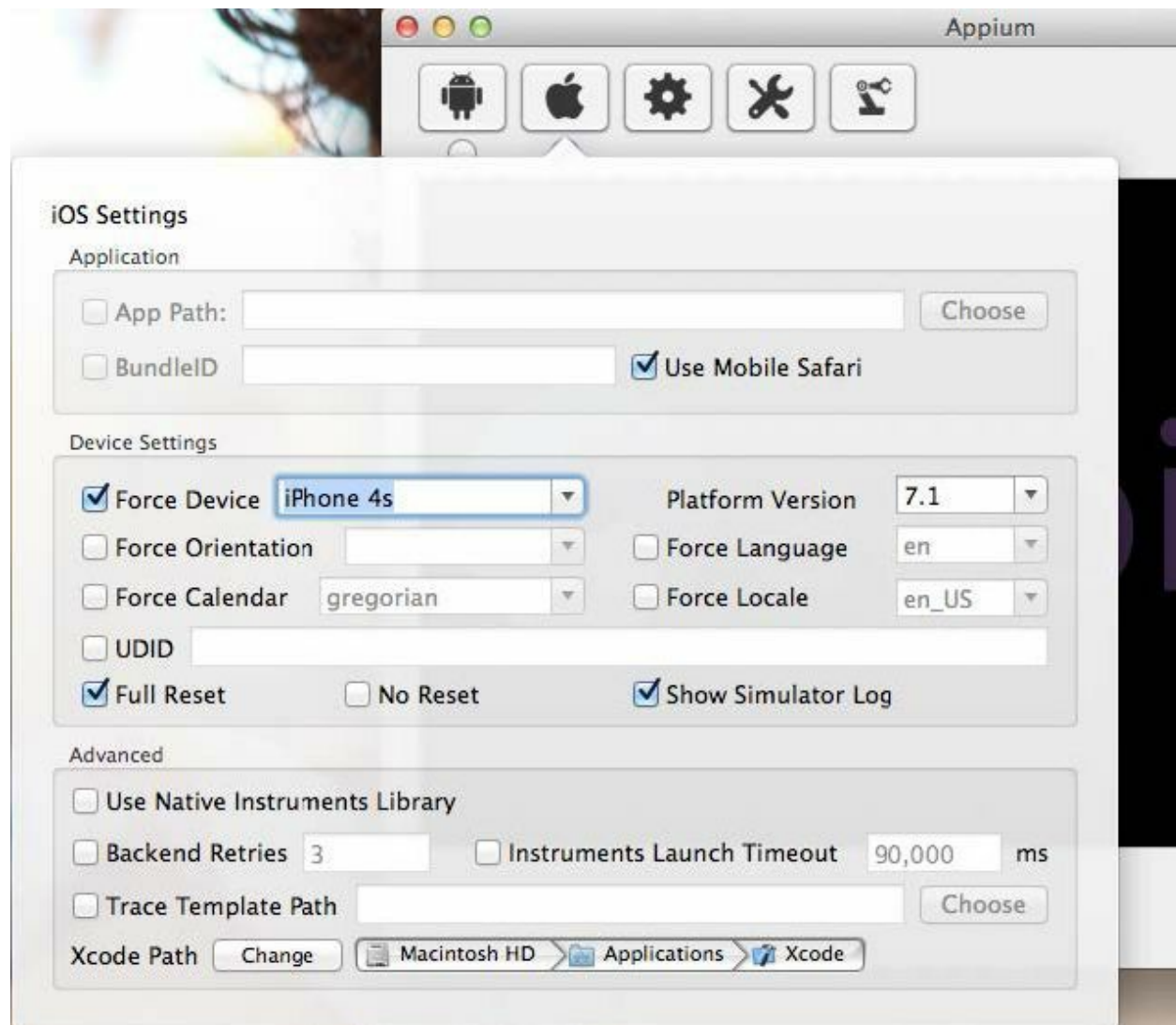
(6) 在Appium主界面，单击Launch按钮启动Appium Server。

Inspector元素定位器：Appium的元素定位工具叫Appium Inspector，可以通过在Appium主界面单击放大镜的图标来启动它。

该定位器提供了定位被测应用的多种分析方式。其中主要的特性就是告诉我们这些UI元素在移动应用程序中是如何被使用的，包括它们的层次结

构以及元素的属性，通过这些我们可以设置元素的定位器。它也可以模拟应用程序上的各种操作手势，并看到在模拟器上执行的效果。它还能够记录下用户在移动应用上的操作步骤。

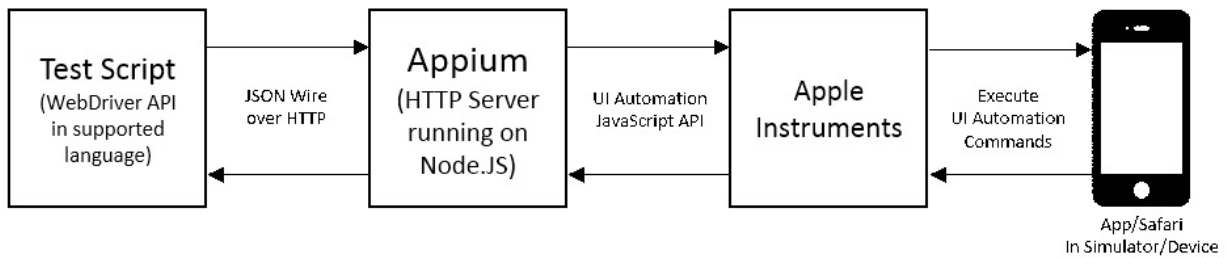




7.3 iOS测试

Appium使用多种自带自动化测试框架来驱动测试执行，并且支持基于Selenium WebDriver JSON无线协议的API调用。对于iOS应用程序的自动化测试，Appium是利用Apple组件之一的UI Automation特性来实现的。

Appium 作为一个HTTP Server来接收测试脚本通过JSON Wire Protocol传输过来的指令。Appium发送这些指令给Apple组件。这样，这些指令就可以在启动了被测应用的模拟器或者真机上执行测试。这个过程中，Appium把JSON指令翻译成UI Automation 能够理解的JavaScript命令。UI Automaiton负责在模拟器或者真机上启动和关闭应用程序。这个过程如下图所示。



当在模拟器或者真机的应用程序上执行测试指令的时候，被测应用程序会发送响应给UI Automation，然后以JavaScript的响应格式返回给Appium。Appium再把UI Automation JavaScript响应翻译成符合Selenium WebDriver JSON Wire Protocol的响应信息，并把它返回给测试脚本。

现在，我们已经成功启动了Appium，接下来创建一个基于iPhone Safari浏览器，检查Web应用搜索功能的测试用例。创建一个新的测试类SearchProductsOnIPhone，代码如下。

```
import unittest
from appium import webdriver

class SearchProductsOnIPhone(unittest.TestCase):
    def setUp(self):
        desired_caps = {}
```

```

# platform
desired_caps['device'] = 'iPhone Simulator'
# platform version
desired_caps['version'] = '7.1'
# mobile browser
desired_caps['app'] = 'safari'

# to connect to Appium server use RemoteWebDrive
r
# and pass desired capabilities
self.driver = \
    webdriver.Remote("http://127.0.0.1:4723/wd/hub", desired_caps)
self.driver.get("http://demo.magentocommerce.com/")

self.driver.implicitly_wait(30)
self.driver.maximize_window()

def test_search_by_category(self):
    # click on search icon
    self.driver.find_element_by_xpath(
        ("//a[@href='#header-search']").click()
    # get the search textbox
    self.search_field = self.driver.find_element_by_name("q")
    self.search_field.clear()
    # enter search keyword and submit
    self.search_field.send_keys("phones")
    self.search_field.submit()

# get all the anchor elements which have product
names
# displayed currently on result page using
# find_elements_by_xpath method
products = self.driver\

```

```
        .find_elements_by_xpath
        ("//div[@class='category-products']/ul/li")

    # check count of products shown in results
    self.assertEqual(2, len(products))

    def tearDown(self):
        # close the browser window
        self.driver.quit()

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

需要使用RemoteWebDriver 来调用Appium执行测试用例。为了在期望的平台上使用Appium，需要执行以下代码。

```
desired_caps = {}
# platform
desired_caps['device'] = 'iPhone Simulator'
# platform version
desired_caps['version'] = '7.1'
# mobile browser
desired_caps['app'] = 'safari'
```

desired_caps['device'] 命令是用Appium来指定在哪个平台上运行测试脚本。上面的例子中，是使用iPhone 模拟器。如果要在iPad上执行测试，那么

就要指定iPad 模拟器。

当在真机上运行测试脚本时，需要为desired_caps['device']赋值为iPhone或者iPad。Appium会选择通过USB连接到Mac电脑上的设备来执行脚本。

desired_caps['version']命令用来设置iPhone/iPad模拟器的版本。上面的例子中，使用的是iOS 7.1版本的模拟器，该版本在写本书的时候是iOS的最新版本。

desired_caps['app']用来设置要启动的目标应用。上面的例子中，启动的是Safari浏览器。

最后，需要通过RemoteWebDriver连接到Appium Server，并且设置好相应的配置。下面是创建一个Remote 实例的代码。

```
self.driver = webdriver.Remote
```

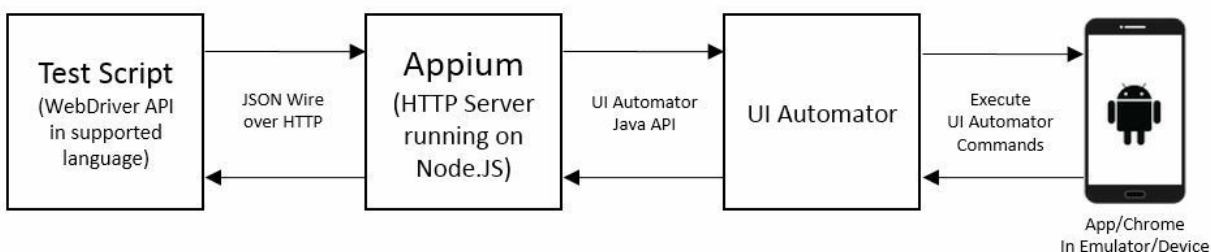
```
("http://127.0.0.1:4723/wd/hub", desired_caps)
```

后面的测试脚本用来调用Selenium API实现与应用的交互操作。运行测试过程中，将会看到Appium通过测试脚本建立会话，并在iPhone模拟器上启动Safari应用程序。Appium将会在模拟器窗口中一步步地执行对于Safari应用的测试。

7.4 Android测试

Appium对于Android应用程序的测试是通过调用内置在Android SDK中的 UI Automator来实现的。过程非常类似于基于iOS平台的测试。

Appium 作为一个HTTP Server来接收测试脚本通过JSON Wire Protocol传输过来的指令。Appium 发送这些指令给UI Automator。这样，这些指令就可以在启动了被测应用的模拟器或者真机上执行测试。这个过程中，Appium把JSON指令翻译成UI Automator 能够理解的Java命令。这个过程如下图所示。



当在模拟器或者真机的应用程序上执行测试指令的时候，被测应用程序会发送响应给UI Automator，然后返回给Appium。Appium再把UI Automator响应翻译成符合Selenium WebDriver JSON Wire Protocol的响应信息，并把它返回给测试脚本。

在Android平台上测试一个应用程序和在iOS平台上的操作非常相似。一般在Android平台，我们会使用真机来取代模拟器（与iOS平台的simulator不同，在Android社区被称为emulator）。下面将使用同样的程序来测试Android平台下的Chrome浏览器。

在本次测试中，将使用Samsung Galaxy S III型号的手机。测试之前首先要安装好Chrome浏览器。Chrome浏览器可以从应用商店下载。然后需要把该手机连接到运行有Appium的电脑上。

下面就要在Android上开始测试工作了。我们是在Android真机上执行测试脚本的，首先要确保手机已经成功安装了Chrome浏览器并且成功连接到电脑上了。运行下面的命令可以获取连接到电脑上的虚拟机和真机的列表。

```
./adb devices
```

Android Debug Bridge(adb) 是内置在Android SDK中的命令行工具，通过它可以与模拟器实例或者真机进行通信。

上面的命令可以获取到已经连接到主机上的Android设备列表。如下图所示，可以看到在上面的例子中已经建立连接的真机。

```
upgundeche — bash — 80x24
Last login: Sun Apr 20 17:26:36 on ttys000
Unmesh-Gundechas-iMac:~ upgundeche$ adb devices
List of devices attached
4df726365fc021e9    device

Unmesh-Gundechas-iMac:~ upgundeche$
```

How to Enable Developer's Options on Android 4.2

As you know, the developer's options are at the core of Android development and as they control the behavior of using an Android device as a development environment. The only change Android 4.2 brings in comparison to these earlier versions is that now they are hidden by default and you will have to make a decision by using some tricks to reveal them.

1. Open Settings App on your Android phone or tablet.
2. If you have a Samsung Galaxy S4, Note 4.2, Tab 3 or any other Galaxy device with Android 4.2, open Settings App, tap About and tap it.
3. If you have a Sony Ericsson or any other device with Android 4.2 go to Settings Menu > About > Developer's Options and tap the Build version 7 times.
4. You will see a toast message and you'll be done.

我们可以修改前面为iOS测试写过的脚本，使其能够在Android平台运行。创建一个新的测试类SearchProductsOnAndroid。复制下面的代码到新创建的测试类中。

```
import unittest
from appium import webdriver

class SearchProductsOnAndroid(unittest.TestCase):
    def setUp(self):
        desired_caps = {}
        # platform
        desired_caps['device'] = 'Android'
        # platform version
```

```

desired_caps['version'] = '4.3'
# mobile browser
desired_caps['app'] = 'Chrome'

# to connect to Appium server use RemoteWebDrive
r
# and pass desired capabilities
self.driver = \
    webdriver.Remote("http://127.0.0.1:4723/wd/hub", desired_caps)
self.driver.get("http://demo.magentocommerce.com/")
self.driver.implicitly_wait(30)

def test_search_by_category(self):

    # click on search icon
    self.driver.find_element_by_xpath
        ("//a[@href='#header-search']").click()
    # get the search textbox
    self.search_field = self.driver.find_element_by_
name("q")
    self.search_field.clear()

    # enter search keyword and submit
    self.search_field.send_keys("phones")
    self.search_field.submit()

    # get all the anchor elements which have product
names
    # displayed currently on result page using
    # find_elements_by_xpath method
    products = self.driver\
        .find_elements_by_xpath
        ("//div[@class='category-products']/ul/li")

```

```
# check count of products shown in results
self.assertEqual(2, len(products))

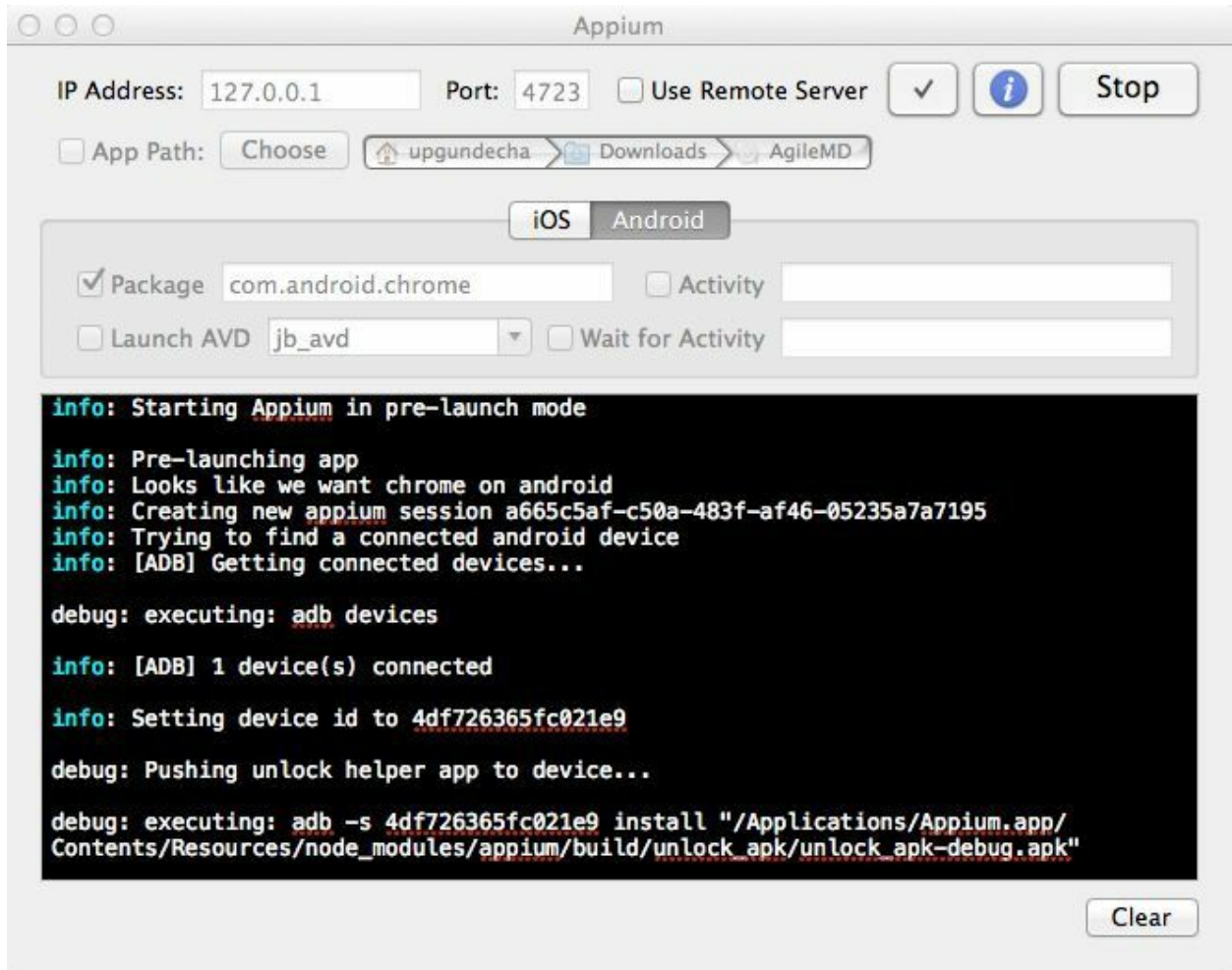
def tearDown(self):
    # close the browser window
    self.driver.quit()

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

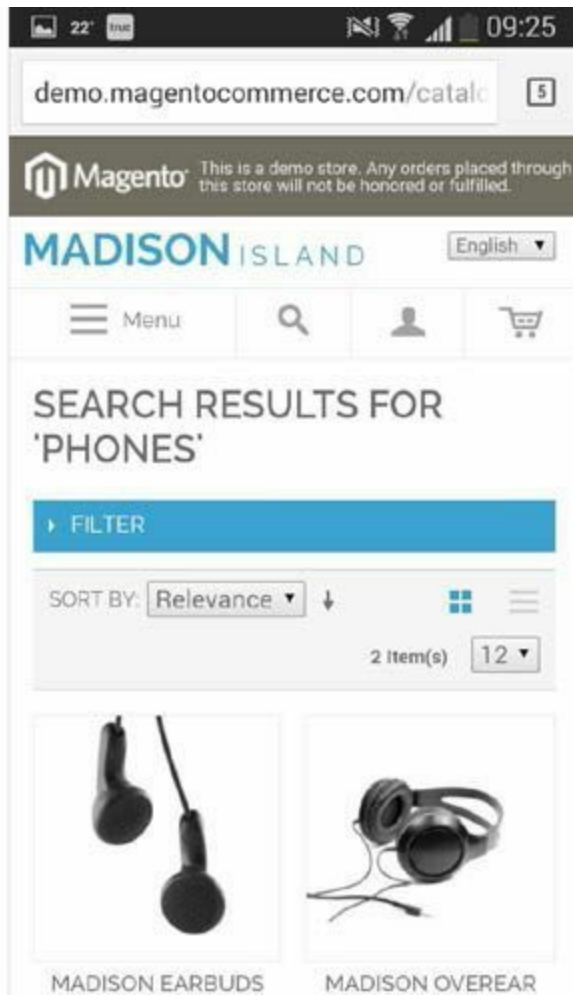
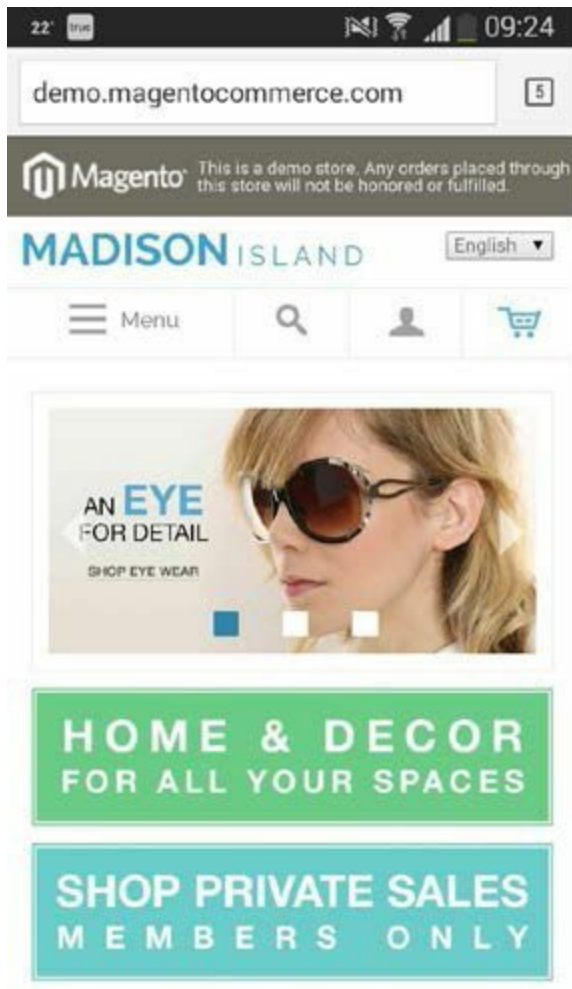
在该实例中，我们给desired_caps['device']指定的值是Android，表明将会在Android平台上运行测试。

接下来，可以看到desired_caps['version']设置的是Android 4.3版本（Jelly Bean）。需要在Android系统的Chrome浏览器上执行测试，因此desired_caps['app']设置的值是Chrome。

Appium将会调用通过adb返回的设备列表中的第一个来运行测试，并通过前面提到的配置，来实现在目标设备上启动Chrome浏览器，然后开始执行测试脚本，如下图所示。



下面是在真机上运行测试脚本时的截图。



7.5 使用Sauce Labs

在第6章讲到跨浏览器测试时，已经提到了Sauce Labs。Sauce也支持通过Appium来测试移动端应用程序。事实上，Appium本身就是由Sauce Labs来开发和支持的项目。通过非常小的配置修改，就可以在Sauce Labs上运行移动端的测试脚本，代码如下。

```
import unittest
from appium import webdriver

class SearchProductsOnIPhone(unittest.TestCase):
    SAUCE_USERNAME = 'upgundecha'
    SUACE_KEY = 'c6e7132c-ae27-4217-b6fa-3cf7df0a7281'

    def setUp(self):
        desired_caps = {}
        desired_caps['browserName'] = "Safari"
        desired_caps['platformVersion'] = "7.1"
        desired_caps['platformName'] = "iOS"
        desired_caps['deviceName'] = "iPhone Simulator"

        sauce_string = self.SAUCE_USERNAME + ':' + self.
SUACE_KEY
```

```

        self.driver = \
            webdriver.Remote('http://' + sauce_string +
                              '@ondemand.saucelabs.com:80/wd/hub', desired
_caps)
        self.driver.get('http://demo.magentocommerce.com
/')
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

    def test_search_by_category(self):
        # click on search icon
        self.driver.find_element_by_xpath("//a[@href=
'#header-search']").click()
        # get the search textbox
        self.search_field = self.driver.find_element_by_
name("q")
        self.search_field.clear()

        # enter search keyword and submit
        self.search_field.send_keys("phones")
        self.search_field.submit()

        # get all the anchor elements which have
        # product names displayed
        # currently on result page using
        # find_elements_by_xpath method
        products = self.driver\
            .find_elements_by_xpath
            ("//div[@class='category-products']/ul/li")

        # check count of products shown in results
        self.assertEqual(2, len(products))

    def tearDown(self):
        # close the browser window
        self.driver.quit()

```

```
if __name__ == '__main__':  
    unittest.main(verbosity=2)
```

当执行完移动端测试以后，可以在Sauce Labs的面板上看到运行的结果和录像回放。Sauce Labs上有很多现成的SDK的组合和设置，在Sauce Labs执行测试可以节省大量用在本地环境配置Appium的时间和精力。

7.6 章节回顾

本章我们学习了如果在移动设备上测试应用程序。我们了解了Appium，它已经成为Selenium用来测试移动端应用程序的一个核心特性。我们完成了Appium的安装和配置，并且通过它测试了本书的样例程序的移动端版本。

我们分别在iPhone模拟器和Android设备上测试移动互联网程序。通过使用Appium，我们可以使用任何一种有WebDriver类库的编程语言来测试各类移动端应用程序。

在下面章节中，将会讲解一些良好的编码实践，比如通过Selenium WebDriver来实现Page Object模式和数据驱动测试。

第8章 Page Object与数据驱动测试

本章将介绍两类重要的设计模式，这些设计模式有助于提升我们自动化测试框架的可扩展性与可维护性。我们将一起学习如何用数据驱动的模式结合Python库去构建Selenium测试脚本。

在本章的第二部分，我们还将学习用Page Object的模式创建高可维护与健壮性的测试脚本。将元素定位器和底层调用从测试脚本中分离出来形成抽象层，如同实现应用程序的各个功能（就像用户在浏览器窗口中体验到的内容一样）。

本章包含以下主题：

- 什么是数据驱动测试；

- 如何用数据驱动的模式（ddt）库结合unittest库构建数据驱动测试；
- 什么是Page Object模式以及如何使用该模式创建维护性好的测试；
- 结合测试样例实现一个Page Object模式的测试。

8.1 数据驱动测试

通过使用数据驱动测试的方法，我们可以在需要验证多组数据的测试场景中，使用外部数据源实现对输入值与期望值的参数化，从而避免在测试中仅使用硬编码的数据。

这种方法对于测试步骤相同而使用不同的“输入值与期望值”的测试场景尤其适用。下表列举了一个用于验证登录场景的数据组合。

场景描述	测试数据	预期结果
有效的用户名和密码	一组有效的用户名和密码	用户正常登录并且收到成功提示信息
无效的用户名和密码	一组无效的用户名和密码	用户收到登录失败的信息
有效的用户名与无效的密码	一个有效的用户名与一个无效的密码	用户收到登录失败的信息

我们只需创建一个测试脚本就可以处理上表的测试数据和条件的各个组合。

使用数据驱动的模式，根据业务逻辑分解测试数据，并且定义变量，使用外部的CSV或表格里的数据使其参数化，从而避免使用原测试脚本中固定的数据。这种方式可以将测试脚本与测试数据分离，使得测试脚本在不同数据集合下高度复用。

数据驱动的模式不仅可以帮助我们增加类似复杂条件场景的测试覆盖，还可以极大地减少我们对测试代码的编写和维护工作量。

接下来的部分，我们将改用Python ddt库，以数据驱动的模式创建前面章节已经实现过的测试。

8.2 使用**ddt**执行数据驱动测试

ddt的库可以将测试中的变量进行参数化。例如，我们可以通过定义一个数组来实现数据驱动测试。

ddt的库包含一组类和方法用于实现数据驱动测试。

8.2.1 安装**ddt**

可以使用下面的命令来下载与安装ddt。

```
pip install ddt
```

更多有关ddt的信息可以访问
<https://pypi.python.org/pypi/ddt>。

8.2.2 设计一个简单的数据驱动测试

我们将一个之前使用数据硬编码的搜索场景测试，转换成用数据驱动模式进行测试，并且使得脚本可以搜索多种类别的商品。

为了创建数据驱动测试，我们需要在测试类上使用@ddt装饰符，在测试方法上使用@data装饰符。@data装饰符把参数当作测试数据，参数可以是单个值、列表、元组、字典。对于列表，需要用@unpack装饰符把元组和列表解析成多个参数。

接下来实现这个搜索测试，传入搜索关键词和期望结果，代码如下。

```
import unittest
from ddt import ddt, data, unpack
from selenium import webdriver

@ddt
class SearchDDT(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()
```

```

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com
/")

    # specify test data using @data decorator
    @data(("phones", 2), ("music", 5))
    @unpack
    def test_search(self, search_value, expected_count)
:
        # get the search textbox
        self.search_field = self.driver.find_element_by_
name("q")
        self.search_field.clear()
        # enter search keyword and submit.
        # use search_value argument to pass data
        self.search_field.send_keys(search_value)
        self.search_field.submit()

        # get all the anchor elements which have
        # product names displayed
        # currently on result page using
        # find_elements_by_xpath method
        products = self.driver.find_elements_by_xpath
            ("//h2[@class='product-name']/a")

        # check count of products shown in results
        self.assertEqual(expected_count, len(products))

    def tearDown(self):
        # close the browser window
        self.driver.quit()

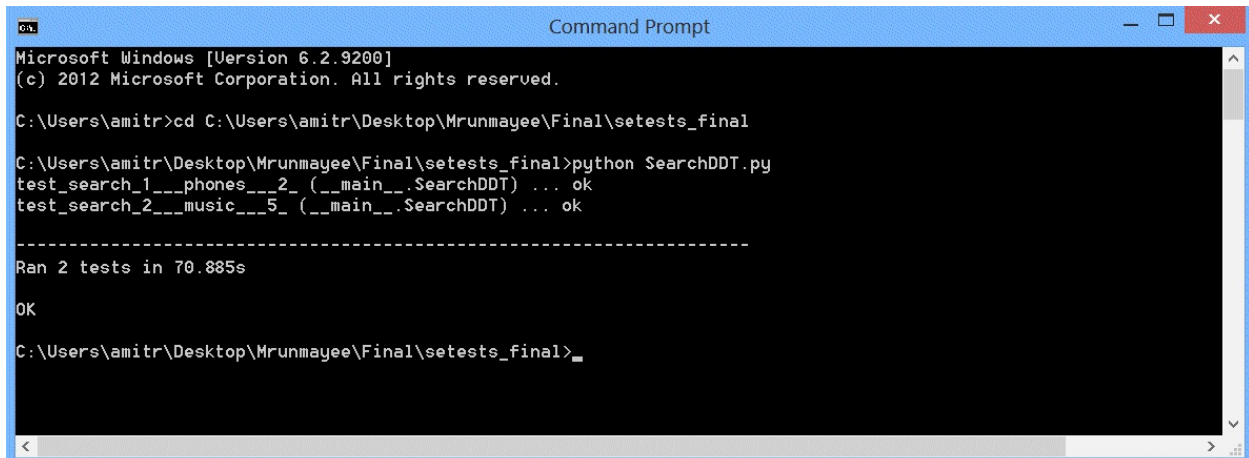
if __name__ == '__main__':
    unittest.main(verbosity=2)

```

在上面的代码里，我们在@data装饰符中把元组列表作为参数，紧接着在@unpack装饰符中把元组解析成多个参数。在test_search()方法中，search_value 与expected_count两个参数用来接收元组解析的数据。

```
# specify test data using @data decorator
@data(("phones", 2), ("music", 5))
@unpack
def test_search(self, search_value, expected_count):
```

当我们运行测试脚本的时候，ddt把测试数据转换为有效的Python标识符，生成名称更有意义的测试方法。例如上面的测试，ddt将生成如下图展示的方法名。



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\amitr>cd C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final

C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final>python SearchDDT.py
test_search_1__phones__2_ (__main__.SearchDDT) ... ok
test_search_2__music__5_ (__main__.SearchDDT) ... ok

-----
Ran 2 tests in 70.885s

OK

C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final>
```

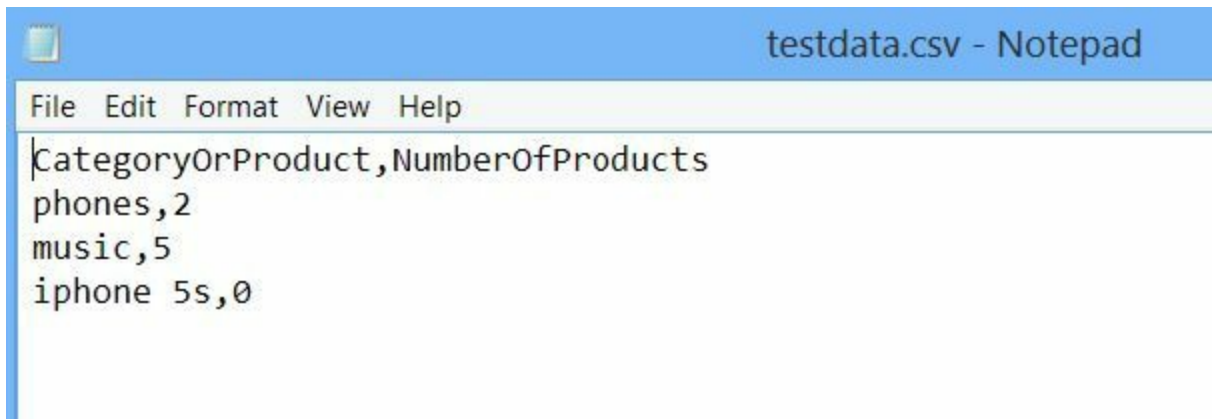
8.3 使用外部数据的数据驱动测试

在先前的例子里，我们在脚本中直接提供了测试数据。然而，有时你会发现所需要的测试数据在测试脚本外部已经存在了，诸如一个文本文件、电子表格或是数据库。这可以使得我们的测试脚本与测试数据分离开来，可以方便我们每次更新与维护测试脚本，而不用担心测试数据。

下面我们一起学习如何借助外部的CSV（逗号分隔值）文件或是Excel表格数据来实现ddt。

8.3.1 通过CSV获取数据

结合前面的测试脚本，我们在@data装饰符中使用解析的外部的CSV（testdata.csv）来换掉之前的测试数据。其中CSV数据文件如下图所示。



接下来，我们在@data装饰符中实现get_data()方法，其中包括路径、CSV文件名。这个方法调用CSV库去读取文件并且返回一行数据。

```
import csv, unittest
from ddt import ddt, data, unpack
from selenium import webdriver

def get_data(file_name):
    # create an empty list to store rows
    rows = []
    # open the CSV file
    data_file = open(file_name, "rb")
    # create a CSV Reader from CSV file
    reader = csv.reader(data_file)
    # skip the headers
    next(reader, None)
    # add rows from reader to list
    for row in reader:
        rows.append(row)
    return rows
```



```

@ddt
class SearchCsvDDT(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com

/")

        # get the data from specified csv file by
        # calling the get_data function
        @data(*get_data("testdata.csv"))
        @unpack
        def test_search(self, search_value, expected_count)
:
            self.search_field =self.driver.find_element_
by_name("q")
            self.search_field.clear()

            # enter search keyword and submit.
            self.search_field.send_keys(search_value)
            self.search_field.submit()

            # get all the anchor elements which have
            # product names displayed
            # currently on result page using
            # find_elements_by_xpath method
            products = self.driver.find_elements_by_xpat
h
                ("//h2[@class='product-name']/a")
            expected_count = int(expected_count)
            if expected_count > 0:
                # check count of products shown in resul

```

```
ts
        self.assertEqual(expected_count, len(products))
    else:
        msg = self.driver.find_element_by_class_name ("note-msg")
        self.assertEqual ("Your search returns no results.", msg.text)

    def tearDown(self):
        # close the browser window
        self.driver.quit()

if __name__ == '__main__':
    unittest.main()
```

测试执行时，@data将调用get_data()方法去读取外部数据文件，并将数据逐行返回给@data。

8.3.2 通过Excel获取数据

用Excel来维护测试数据是最常用的做法。这还可以帮助非技术人员很轻松地添加一行需要的测试数据。结合上面的例子，我们把数据整理到Excel中，如下图所示。

A	B
Category/Product	NumberOfProducts
phones	2
music	5
iphone 5s	0

读取Excel文件，我们需要用到另外一个叫xlrd的库，其安装命令如下。

```
pip install xlrd
```



xlrd库提供了读取工作簿、工作表以及单元格的方法。如果需要往表格中写数据，则需要用到xlwt库。另外，openpyxl提供了对电子表格可读可写的功能。

接下来我们修改get_data()方法，试着从外部的电子表格获取测试数据，代码如下。

```
import xlrd, unittest
from ddt import ddt, data, unpack
from selenium import webdriver
```

```

def get_data(file_name):
    # create an empty list to store rows
    rows = []
    # open the specified Excel spreadsheet as workbook
    book = xlrd.open_workbook(file_name)
    # get the first sheet
    sheet = book.sheet_by_index(0)
    # iterate through the sheet and get data from rows
in list
    for row_idx in range(1, sheet.nrows):
        rows.append(list(sheet.row_values(row_idx, 0, sheet.ncols)))
    return rows

@ddt
class SearchExcelDDT(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com/")

        # get the data from specified Excel spreadsheet
        # by calling the get_data function
        @data(*get_data("TestData.xlsx"))
        @unpack
        def test_search(self, search_value, expected_count)
:
            self.search_field = self.driver.find_element
_by_name("q")
            self.search_field.clear()

```

```

        # enter search keyword and submit.
        self.search_field.send_keys(search_value)
        self.search_field.submit()

        # get all the anchor elements which have
        # product names displayed
        # currently on result page using
        # find_elements_by_xpath method
        products = self.driver.find_elements_by_xpat
h
            ("//h2[@class='product-name']/a")
        if expected_count > 0:
            # check count of products shown in resul
ts
                self.assertEqual(expected_count, len(pro
ducts))
        else:
            msg = self.driver.find_element_by_class
_name("note-msg")
            self.assertEqual("Your search returns no
results.", msg.text)

        def tearDown(self):
            # close the browser window
            self.driver.quit()

if __name__ == '__main__':
    unittest.main()

```

类似读取CSV文件一样，@data将调用
get_data()方法去读取外部Excel文件，并将数据逐

行返回给@data。



通过数据库获取数据

如果你想要从数据库的库表中获取数据，那么你同样需要修改`get_data()`方法，并且通过DB相关的库来连接数据库、SQL查询来获取测试数据。

8.4 Page Object设计模式

到目前为止，我们已经掌握了在Python unittest单元测试框架中，编写Selenium WebDriver测试脚本的方法，并且可以在测试类中根据测试用例的步骤使用合适的定位器。然后，随着陆续越来越多的测试场景加入自动化测试用例，那么与之对应的测试脚本就变得越来越难以维护，甚至代码变得很脆弱。

开发可维护性和可重用的测试脚本，对于持续自动化测试是很重要的。其重要性完全不亚于我们被测产品的代码标准。

如何去解决这些问题呢？如果你是一个开发工程师，你可能会调动很多原则和方法，诸如Don't Repeat Yourself（DRY，高级码农的信条，不做重复的自己，不写重复的代码）或定期代码重构的方

法。Page Object模式，是使用Selenium的广大同行最为公认的一种设计模式。在设计测试时，把元素和方法按照页面抽象出来，分离成一定的对象，然后再进行组织。它相较之前的Facade模式（为系统中的一组接口所提供的一个一致的界面）又更进了一步。

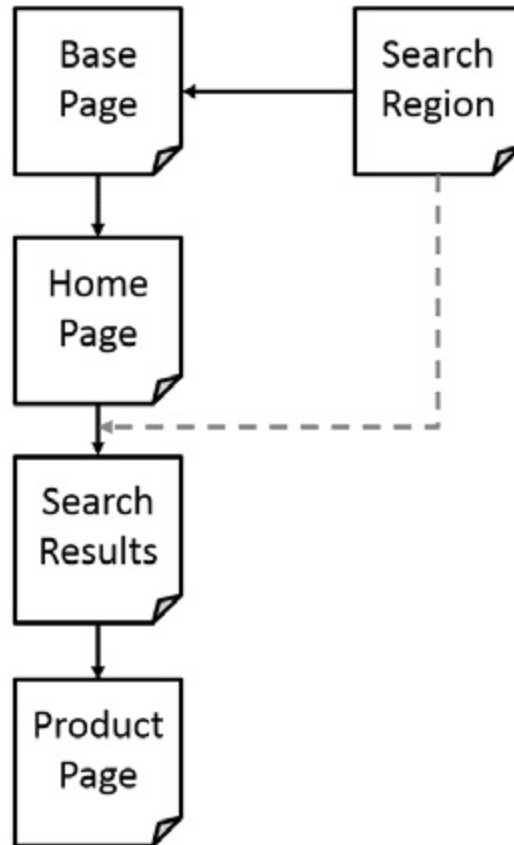
Page Object模式，创建一个对象来对应页面的一个应用。因此，我们可以为每个页面定义一个类，并为每个页面的属性和操作构建模型。这就相当于在测试脚本和被测的页面功能中分离出一层，屏蔽了定位器、底层处理元素的方法和业务逻辑，取而代之的是，Page Object会提供一系列的API来处理页面功能。

测试应该在更上层使用这些页面对象，在底层页面中的属性或操作的任何更改都不会中断测试。Page Object模式具有以下几个优点。

- 抽象出对象可以最大程度地降低开发人员修改页面代码对测试的影响，所以，你仅需要对页面对象进行调整，而对测试没有影响；
- 可以在多个测试用例中复用一部分测试代码；
- 测试代码变得更易读、灵活、可维护。

接下来，我们一起将前面章节已经实现过的测试脚本进行重构，对原应用程序页面抽象出对象。在这个示例中，我们将针对被测页面创建如下图所示的结构。首先我们要实现一个Base Page 对象

（可以理解为其他页面所要用到的模板），对应的Base 对象提供了其他页面需要用到的功能区块。例如，所有的页面都用到搜索功能，那么我们可以基于Base Page创建一个Search对象，接下来我们分别为首页、搜索结果页以及产品页面各自创建类。



8.4.1 测试准备

在我们开始用Page Object模式设计测试之前，首先得先实现一个名为BaseTestCase的类，用于给我们提供setUp()和tearDown()两种方法，以便后续我们写每个类都可以拿来复用。创建名为basetestcase.py的脚本，代码细节如下。

```
import unittest
from selenium import webdriver
```

```
class BaseTestCase(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get('http://demo.magentocommerce.com
/')

    def tearDown(self):
        # close the browser window
        self.driver.quit()
```

8.4.2 BasePage对象

BasePage 对象相当于所有页面对象中的父对象，同时可以提供公共部分的代码。创建名为 base.py 的脚本，代码细节如下。

```
from abc import abstractmethod
class BasePage(object):
    """ All page objects inherit from this """

    def __init__(self, driver):
        self._validate_page(driver)
        self.driver = driver

    @abstractmethod
```

```
def _validate_page(self, driver):
    return

    """ Regions define functionality available through
all page objects """
    @property
    def search(self):
        from search import SearchRegion
        return SearchRegion(self.driver)

class InvalidPageException(Exception):
    """ Throw this exception when you don't find the co
rrect page """
    pass
```

我们增加了一个名为`_validate_page()`的抽象方法，继承`BasePage`的`page`对象将实现这个方法，目的是在能够使用属性和操作之前，验证页面是否已经加载到浏览器。

另外，我们还创建了`search`属性用于返回`SearchRegion`对象。类似于一个页面对象，`SearchRegion`相当于每个页面都用到的搜索框。所以接下来其他页面对象都可以共享这个`BasePage`类。

最后，我们也实现了`_validate_page()`方法中用到的`InvalidPageException`，如果页面验证失败，`InvalidPageException`将会被抛出。

8.4.3 实现Page Object

现在，我们可以为每个页面实现Page Object了。

(1) 首先，我们定义`HomePage`，创建`homepage.py`，实现`HomePage`类，代码如下。

```
from base import BasePage
from base import InvalidPageException

class HomePage(BasePage):

    _home_page_slideshow_locator = 'div.slideshow-container'

    def __init__(self, driver):
        super(HomePage, self).__init__(driver)

    def _validate_page(self, driver):
        try:
            driver.find_element_by_class_name (self._home_page_slideshow_locator)
```

```
except:
    raise InvalidPageException ("Home Page not loaded")
```

我们要遵循的一点是将定位器字符串与它们的使用位置分离开。我们可以创建一个“_”前缀的私有变量，例如，用_home_page_slideshow_locator变量来保存应用程序首页的slideshow组件的定位器字符串。我们可以利用这个来确认浏览器是否正常加载了首页。

```
_home_page_slideshow_locator = 'div.slideshow-container'
```

然后，我们可以创建_validate_page()方法，通过判断slideshow元素是否已经显示在首页上了，来判断首页是否被加载。

(2) 接下来，我们实现SearchRegion类，包括searchFor()方法，该方法用于返回SearchResults类对应的搜索结果页面。创建一个新的脚本search.py，并且实现这两个类，代码如下。

```

from base import BasePage
from base import InvalidPageException
from product import ProductPage

class SearchRegion(BasePage):
    _search_box_locator = 'q'

    def __init__(self, driver):
        super(SearchRegion, self).__init__(driver)

    def searchFor(self, term):
        self.search_field = self.driver.find_element_by_
name (self._search_box_locator)
        self.search_field.clear()
        self.search_field.send_keys(term)
        self.search_field.submit()
        return SearchResults(self.driver)

class SearchResults(BasePage):
    _product_list_locator = 'ul.products-grid > li'
    _product_name_locator = 'h2.product-name a'
    _product_image_link = 'a.product-image'
    _page_title_locator = 'div.page-title'

    _products_count = 0
    _products = {}

    def __init__(self, driver):
        super(SearchResults, self).__init__(driver)
        results = self.driver.find_elements_by_css_selec
tor (self._product_list_locator)
        for product in results:
            name = product.find_element_by_css_selector
(self._product_name_locator).text
            self._products[name] = product.find_element_by_c

```

```
ss_selector (self._product_image_link)

def _validate_page(self, driver):
    if 'Search results for' not in driver.title:
        raise InvalidPageException ('Search results not
loaded')

@property
def product_count(self):
    return len(self._products)

def get_products(self):
    return self._products

def open_product_page(self, product_name):
    self._products[product_name].click()
    return ProductPage(self.driver)
```

(3) 最后，我们要实现ProductPage类，这个类包括了很多有关商品的一些属性。访问一个商品的详细页面，可以通过SearchResults类，打开搜索结果中一个具体的产品。创建product.py脚本文件实现ProductPage类，代码如下。

```
from base import BasePage
from base import InvalidPageException

class ProductPage(BasePage):
    _product_view_locator = 'div.product-view'
    _product_name_locator = 'div.product-name'
```



```

span'
    _product_description_locator    = 'div.tab-content d
iv.std'
    _product_stock_status_locator   = 'p.availability sp
an.value'
    _product_price_locator          = 'span.price'

    def __init__(self, driver):
        super(ProductPage, self).__init__(driver)

    @property
    def name(self):
        return self.driver.\
            find_element_by_css_selector
            (self._product_name_locator)\
            .text.strip()

    @property
    def description(self):
        return self.driver.\
            find_element_by_css_selector
            (self._product_description_locator)\
            .text.strip()

    @property
    def stock_status(self):
        return self.driver.\
            find_element_by_css_selector
            (self._product_stock_status_locator)\
            .text.strip()

    @property
    def price(self):
        return self.driver.\
            find_element_by_css_selector
            (self._product_price_locator)\

```

```
        .text.strip()

    def _validate_page(self, driver):
        try:
            driver.find_element_by_css_selector (self._product_view_locator)
        except:
            raise InvalidPageException ('Product page not loaded')
```

你还可以进一步添加一些测试场景，例如在商品页点击商品添加到购物车，或者比较商品，又或者通过商品属性返回相关的商品。

8.4.4 构建Page Object模式测试实例

结合之前的准备，我们可以构建完整测试了。下面我们创建一个用于检验应用程序搜索功能的测试，使用BaseTestCase类并调用我们之前创建的页面对象。该测试首先创建一个HomePage实例，并调用searchFor()方法返回SearchResults实例。然后调用SearchResults类中的open_product_page()方法，来打开返回的搜索结果中商品的详情页，进而检查商

品属性。

相关脚本searchtest.py，以及其中SearchProductTest测试类代码如下。

```
import unittest
from homepage import HomePage
from BaseTestCase import BaseTestCase

class SearchProductTest(BaseTestCase):
    def testSearchForProduct(self):
        homepage = HomePage(self.driver)
        search_results = homepage.search.searchFor('earp
hones')
        self.assertEqual(2, search_results.product_count
)
        product = search_results.open_product_page ('MAD
ISON EARBUDS')
        self.assertEqual('MADISON EARBUDS', product.name
)
        self.assertEqual('$35.00', product.price)
        self.assertEqual('IN STOCK', product.stock_statu
s)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

注意，在这里我们并没有写setUp()和tearDown()两个方法。我们只需要继承BaseTestCase

类，直接使用其已经实现过的方法即可。如果你有特殊的测试场景，当然也可以重载这两个方法。

通过对上述例子完整的学习，我们已经掌握了一个页面完整工作流的Page Object设计测试的实践。你也可以通过类似的模式对如购物车、账号注册、登录等场景设计测试了。

8.5 章节回顾

本章我们认识了编写数据驱动测试的方法，以及构建可复用、易量化、维护性好的Page Object模式测试脚本。其中数据驱动的方法，可以将我们的测试数据与测试脚本分离开来，使得我们可以使用更复杂的测试数据，而不用编写新的测试脚本。结合unittest和ddt库，我们可以轻松实现多种外部数据源的测试数据获取。另外，关于如何用Page Object模式设计测试脚本，以及如何对一个简单的业务场景设计一个高可维护性的测试脚本做了详细的介绍。

在接下来的章节，我们将学习Selenium WebDriver API的一些高级特性，例如常用的截屏、屏幕录制、模拟鼠标与键盘操作、操作cookies等。

第9章 Selenium WebDriver的高级特性

到目前为止，我们已经学习了如何使用 Selenium WebDriver来测试Web应用，以及如何通过WebDriver中的一些主要的接口与页面元素进行交互。

在本章，我们将进一步探讨WebDriver中的一些高级API，当测试较复杂的应用场景时，这些功能将派上用场。

本章包含以下主题：

- 通过Action类模拟键盘或鼠标事件；
- 模拟一些鼠标操作，例如拖拽、双击等；
- 调用JavaScript；

- 截屏与录制;
- 处理导航与cookies;
- 键盘与鼠标事件。

9.1 键盘与鼠标事件

WebDriver高级应用的API，允许我们模拟简单到复杂的键盘和鼠标事件，如拖拽操作、快捷键组合、长按以及鼠标右键操作。这些都是通过使用WebDriver的Python API中ActionChains类实现的。

下表列出ActionChains类中一些关于键盘和鼠标事件的重要方法。

方 法	描 述	参 数	样 例
<code>click(on_element=None)</code>	单击元素操作	<code>on_element:</code> 指被单击的元素。如果该参数为None，将单击当前鼠标位置	<code>click(main_link)</code>
<code>click_and_hold(on_element=None)</code>	对元素按住鼠标	<code>on_element:</code> 指被单击且按住鼠标左键的元素。如果该	<code>click_and_hold(gmail_link)</code>

	标左键	参数为 None，将单 击当前鼠标位 置	
double_click(on_element=None)	双击元 素操作	on_element: 指被双击的元 素。如果该参 数为None， 将双击当前鼠 标位置	double_click(info_box)
drag_and_drop(source, target)	鼠标拖 动	source: 鼠标拖动的源 元素。 target: 鼠标释放的目 标元素	drag_and_drop(img, canvas)
key_down(value, element=None)	仅按下 某个 键，而 不释 放。这 个方法 用于修 饰键 (如 Ctrl、 Alt与 Shift 键)	key:指修饰 键。Key的值 在Keys类中定 义。 target: 按键 触发的目标元 素，如果为 None，则按 键在当前鼠标 聚焦的元素上 触发	key_down(Keys. SHIFT)\

			send_keys('n')\nkey_up(Keys.SHIFT)
key_up(value, element=None)	用于释放修饰键	key:指修饰键。Key的值在Keys类中定义。 target: 按键触发的目标元素，如果为None，则按键在当前鼠标聚焦的元素上触发	
move_to_element(to_element)	将鼠标移动至指定元素的中央	to_element: 指定的元素	move_to_element(gmail_link)
perform()	提交（重放）已保存的动作		perform()
release(on_element=None)	释放鼠标	on_element: 被鼠标释放的元素	release(banner_img)
send_keys(keys_to_send)	对当前焦点元素的键	keys_to_send: 键盘的输入值	send_keys("hello")

	盘操作		
<code>send_keys_to_element(element, keys_to_send)</code>	对指定元素的键盘操作	element: 指定的元素。 keys_to_send: 键盘的输入值	<code>send_keys_to_element(firstName, "John")</code>

获取更多细节可访问

http://selenium.googlecode.com/git/docs/api/py/webdriver/webdriver.common.action_chains.html。



Interactions API目前不支持Safari浏览器，同时在其他浏览器上也有一定的局限性。更多详情请访问

<https://code.google.com/p/selenium/wiki/AdvancedUserInteractions>。

9.1.1 键盘事件

接下来我们创建一个测试脚本，用来模拟一个组合键的操作。在这个简单的场景中，当我们按下 Shift+N 组合键时，label 标签会改变颜色。代码如下。

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys
import unittest

class HotkeyTest(unittest.TestCase):
    URL = "https://rawgit.com/jeresig/jquery.hotkeys/master/test-static-05.html"

    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.get(self.URL)
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

    def test_hotkey(self):
        driver = self.driver

        shift_n_label = WebDriverWait(self.driver, 10).\
```

```

        until(expected_conditions.visibility_of_element_
            located((By.ID, "_shift_n")))

        ActionChains(driver).\
            key_down(Keys.SHIFT).\
            send_keys('n').\
            key_up(Keys.SHIFT).perform()
        self.assertEqual("rgba(12, 162, 255, 1)",
            shift_n_label.value_of_css_
                property("background-color"))

    def tearDown(self):
        self.driver.close()

if __name__ == "__main__":
    unittest.main(verbosity=2)

```

通过使用ActionChains类，我们可以实现组合键操作。在上面的示例中，我们联合key_down()、send_key()与key_up()三个方法模拟真人操作Shift+N组合键。

```

ActionChains(driver).\
    key_down(Keys.SHIFT).\
    send_keys('n').\
    key_up(Keys.SHIFT).perform()

```

当调用ActionChains类的方法时，它不会立即

执行，而是会将所有的操作按顺序存放在一个队列里，当调用`perform()`方法时，队列中的事件会依次执行。

9.1.2 鼠标事件

下面演示一个调用`ActionChains`类中的`move_to_element()`方法实现鼠标移动的示例。这个方法类似于`onMouseOver`事件。`move_to_element()`方法是将光标从当前位置移动到指定的元素。

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions
from selenium.webdriver.common.action_chains import ActionChains
import unittest

class ToolTipTest (unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.get("http://jqueryui.com/tooltip/")
        self.driver.implicitly_wait(30)
```

```

        self.driver.maximize_window()

    def test_tool_tip(self):
        driver = self.driver

        frame_elm = driver.find_element_by_class_name("demo-frame")
        driver.switch_to.frame(frame_elm)

        age_field = driver.find_element_by_id("age")
        ActionChains(self.driver).move_to_element(age_field).perform()

        tool_tip_elm = WebDriverWait(self.driver, 10)\
            .until(expected_conditions.visibility_of_element_
                located((By.CLASS_NAME, "ui-tooltip-content"
                    )))

        # verify tooltip message
        self.assertEqual("We ask for your age only for statistical
            purposes.", tool_tip_elm.text)

    def tearDown(self):
        self.driver.close()

if __name__ == "__main__":
    unittest.main(verbosity=2)

```

9.1.2.1 双击操作

调用ActionChains类中的double_click()方法实

现鼠标对元素的双击操作，代码如下。

```
from selenium import webdriver

from selenium.webdriver.common.action_chains import ActionChains
import unittest

class DoubleClickTest (unittest.TestCase):
    URL = "http://api.jquery.com/dblclick/"

    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.get(self.URL)
        self.driver.maximize_window()

    def test_double_click(self):
        driver = self.driver
        frame = driver.find_element_by_tag_name("iframe"
)
        driver.switch_to.frame(frame)
        box = driver.find_element_by_tag_name("div")

        # verify color is Blue
        self.assertEqual("rgba(0, 0, 255, 1)",
                        box.value_of_css_property("background-color"))

        ActionChains(driver).move_to_element
            ( driver.find_element_by_tag_name("span"))\
            .perform()

        ActionChains(driver).double_click(box).perform()
```



```
# verify Color is Yellow
self.assertEqual("rgba(255, 255, 0, 1)",
                  box.value_of_css_property("background-color"))

def tearDown(self):
    self.driver.close()

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

9.1.2.2 鼠标拖动

调用ActionChains类中的drag_and_drop()方法实现鼠标的拖放操作。这个方法拖动源元素，然后在目标元素的位置释放源元素。示例如下。

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
import unittest

class DragAndDropTest (unittest.TestCase):

    URL = "http://jqueryui.com/resources/demos/droppable/default.html"

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.get(self.URL)
```

```
        self.driver.maximize_window(30)
        self.driver.maximize_window()

    def test_drag_and_drop(self):
        driver = self.driver

        source = driver.find_element_by_id("draggable")
        target = driver.find_element_by_id("droppable")

        ActionChains(self.driver).drag_and_drop(source,
target).perform()
        self.assertEqual("Dropped!", target.text)

    def tearDown(self):
        self.driver.close()

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

9.2 调用JavaScript

在执行某些特殊操作或测试JavaScript代码时，WebDriver还提供了调用JavaScript的方法。WebDriver类包含的相关方法见下表。

方 法	描 述	参 数	示 例
<code>execute_async_script(script, *args)</code>	异步执行JS代码	script:被执行的JS代码。 args:JS代码中的任意参数	<code>driver.execute_async_script("return document.title")</code>
<code>execute_script(script, *args)</code>	同步执行JS代码	script:被执行的JS代码。 args:JS代码中的任意参数	<code>driver.execute_script("return document.title")</code>

接下来创建的测试用到了工具方法，该工具方法在使用JavaScript方法对元素执行操作之前，先对它们进行高亮显示。

```
from selenium import webdriver
import unittest
```

```

class ExecuteJavaScriptTest (unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com
/")

    def test_search_by_category(self):

        # get the search textbox
        search_field = self.driver.find_element_by_name(
"q")

        self.highlightElement(search_field)
        search_field.clear()

        # enter search keyword and submit
        self.highlightElement(search_field)
        search_field.send_keys("phones")
        search_field.submit()

        # get all the anchor elements which have product
names
        # displayed currently on result page using
        # find_elements_by_xpath method
        products = self.driver.find_elements_by_xpath("/
/h2[@ class='product-name']/a")

        # check count of products shown in results
        self.assertEqual(2, len(products))

```

```
def tearDown(self):
    # close the browser window
    self.driver.quit()

def highlightElement(self, element):
    self.driver.execute_script("arguments[0].setAttribute('style',
        arguments[1]);",
        element, "color: green;
        border: 2px solid green;")
    self.driver.execute_script("arguments[0].setAttribute('style',
        arguments[1]);",
        element, "")

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

我们可以通过调用WebDriver类的execute_script方法来执行JavaScript代码，也可以通过这个方法传递参数给JavaScript代码，示例代码如下。在这个例子中，我们修改了边框样式，然后又立即恢复到原来的样子。在执行期间，元素将以绿色边框高亮显示，这对于了解哪一个步骤正在执行是非常有用的。

```
def highlightElement(self, element):
    self.driver.execute_script("arguments[0].setAttrib
```

```
ute('style',
    arguments[1]);",
    element, "color: green; border: 2px solid green;")
    self.driver.execute_script("arguments[0].setAttrib
ute('style',
    arguments[1]);",
    element , "")
```

9.3 屏幕截图

自动测试执行过程中，在出错时捕获屏幕截图，是我们在跟开发人员探讨错误时的重要依据。WebDriver内置了一些在测试执行过程中捕获屏幕并保存的方法，如下表所示。

方 法	描 述	参 数	示 例
save_screenshot(filename)	获取当前屏幕截图并保存为指定文件	filename: 指定保存的路径/图片文件名	Driver.save_screenshot ("homepage.png")
get_screenshot_as_base64()	获取当前屏幕截图 base64编码字符串（用于HTML页面直接嵌入base64编码图片）		driver.get_screenshot_as_base64()
	获取当前的屏幕截图，使用完整的路径。如果	filename: 指定保存	driver.get_

get_screenshot_as_file(filename)	有任何 IOError，返 回False，否 则返回True	的路径/图 片文件名	screenshot_as_file('/results/ screenshots/ HomePage.png')
get_screenshot_as_png()	获取当前屏 幕截图的二 进制文件数 据		driver.get_screenshot_as_png()

接下来，我们通过屏幕截图来捕获一个测试执行出错的场景。场景中，我们定位一个本来应该显示在主页的元素。如果测试脚本没有发现对应元素，则立即抛出NoSuchElementException异常，同时截取当前浏览器窗口截图，我们可以把它作为bug的依据发给开发人员定位问题。

```
from selenium import webdriver
import datetime, time, unittest
from selenium.common.exceptions import NoSuchElementException

class ScreenShotTest(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.get("http://demo.magentocommerce.com
```



```

/")

    def test_screen_shot(self):
        driver = self.driver
        try:
            promo_banner_elem = driver.find_element_by_id("promo_banner")
            self.assertEqual("Promotions", promo_banner_elem.text)
        except NoSuchElementException:
            st = datetime.datetime\
                .fromtimestamp(time.time()).strftime('%Y%m%d_%H%M%S')
            file_name = "main_page_missing_banner" + st + ".png"
            driver.save_screenshot(file_name)
            raise

    def tearDown(self):
        self.driver.close()

if __name__ == "__main__":
    unittest.main(verbosity=2)

```

在上述代码中，当测试脚本找不到“promo_banner”元素时，程序就调用save_screenshot()方法来自动截屏，并以我们定义的图片文件名保存在指定的路径下。

```

try:
    promo_banner_elem = driver.find_element_by_id("prom

```

```
o_banner")
    self.assertEqual("Promotions", promo_banner_elem.text)
except NoSuchElementException:
    st = datetime.datetime.fromtimestamp(time.time()).
    strftime('%Y%m%d_%H%M%S')
    file_name = "main_page_missing_banner" + st + ".png"
    "
    driver.save_screenshot(file__name)
    raise
```



当我们使用上述截屏方法时，推荐使用包含唯一标识（例如时间戳）的名称，并且保存为PNG图片等高压缩图片格式，来控制图片的大小。

9.4 屏幕录制

类似屏幕截图，屏幕录制能够更好地帮助我们记录测试过程中到底发生了什么。录像材料可以作为提交问题时的依据发送给项目相关人员，也可以作为产品的功能演示。

然而，Selenium WebDriver没有内置录制的功能，所以要依赖Python类库中名为Castro的工具。这是由Selenium创始人Jason Huggin设计的。Castro是基于跨平台屏幕录制工具Pyvnc2swf开发的。它使用VNC协议录制屏幕并生成SWF视频文件。

由于符合VNC协议，所以我们还可以实现对远程机器（预装VNC相关程序包）的屏幕录制。先安装PyGame，然后安装Castro，pip命令如下。

```
pip install Castro
```

如果Server和Viewer端都是Windows的环境，我们可以选择安装**TightVNC**工具。

如果在Ubuntu操作系统上，可以依次操作**Settings | Preference | Remote Desktop**，然后选中**Allow other users to view your desktop**复选框。在Mac上，我们可以安装Vine VNC Server或者在**System Preferences**中打开**Remote Desktop**。

结合之前章节我们设计过的测试脚本，添加屏幕录制功能，代码如下。

```
import unittest
from selenium import webdriver
from castro import Castro

class SearchProductTest(unittest.TestCase):
    def setUp(self):
        # create an instance of Castro and provide name
        # for the output
        # file
        self.screenCapture = Castro(filename="testSearch
ByCategory. swf")
        # start the recording of movie
        self.screenCapture.start()
```

```

# create a new Firefox session
self.driver = webdriver.Firefox()
self.driver.implicitly_wait(30)
self.driver.maximize_window()

# navigate to the application home page
self.driver.get("http://demo.magentocommerce.com
/")

def test_search_by_category(self):

    # get the search textbox
    search_field = self.driver.find_element_by_name(
"q")
    search_field.clear()

    # enter search keyword and submit
    search_field.send_keys("phones")
    search_field.submit()

    # get all the anchor elements which have product
names
    # displayed
    # currently on result page using find_elements_b
y_xpath method
    products = self.driver.find_elements_by_xpath("/
/h2[@
class='product-name']/a")

    # check count of products shown in results
    self.assertEqual(2, len(products))

def tearDown(self):
    # close the browser window
    self.driver.quit()
    # Stop the recording

```

```
self.screenCapture.stop()

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

从代码中可以看到，要创建一个录制屏幕的会话，我们需要创建一个Castro对象并且使用录像文件的路径和名称作为参数初始化实例。start()和stop()方法用于控制屏幕录制的起止位。代码中setUp()方法的部分就是一个最佳的初始化Castro实例，并且是开始录制的示例。

```
def setUp(self):
    # Create an instance of Castro and provide name for
    # the output
    # file
    self.screenCapture = Castro(filename="testSearchByCategory.swf")
    # Start the recording of movie
    self.screenCapture.start()

    # create a new Firefox session
    self.driver = webdriver.Firefox()
    self.driver.implicitly_wait(30)
    self.driver.maximize_window()

    # navigate to the application home page
    self.driver.get("http://demo.magentocommerce.com/")
```

在teadDown()部分，我们可以看到当完整的测试用例都执行完成后，调用stop()方法来停止屏幕录制。代码如下。

```
def tearDown(self):  
    # close the browser window  
    self.driver.quit()  
    # Stop the recording  
    self.screenCapture.stop()
```

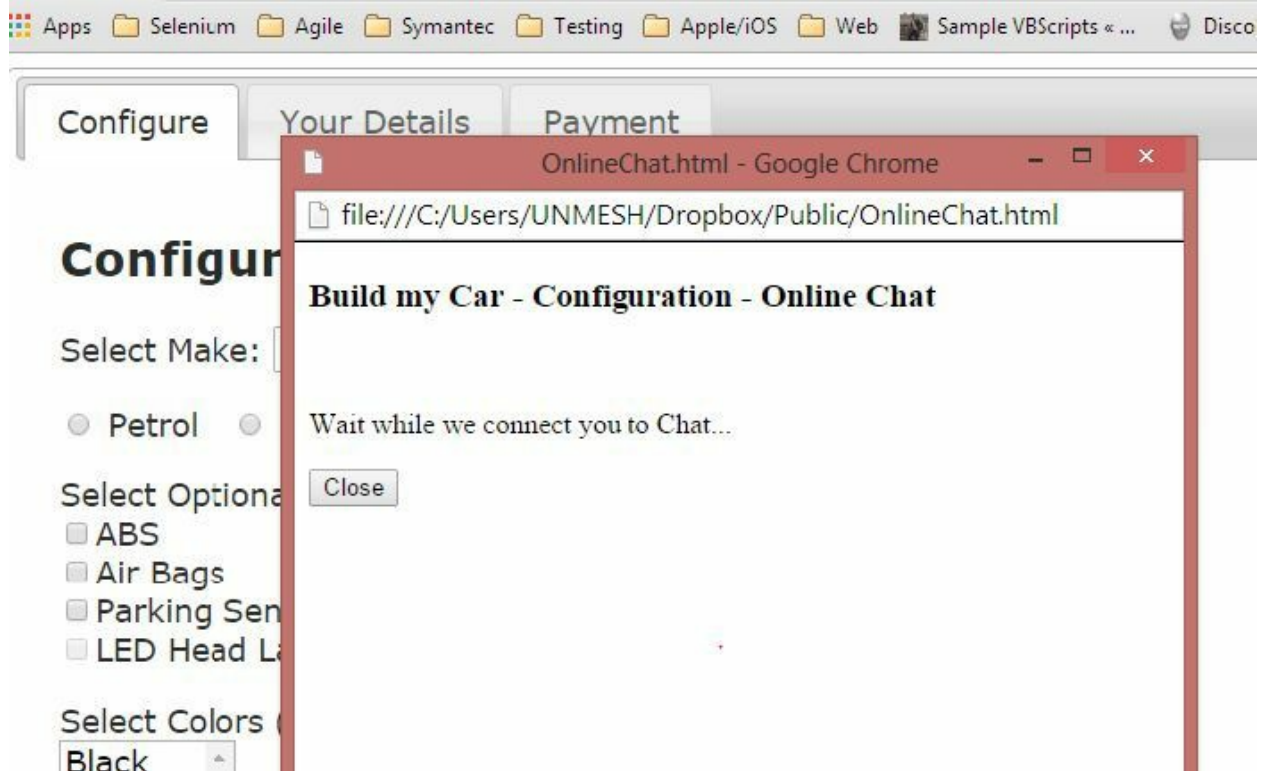
特别是在组合多个测试场景的测试类中，我们同样也可用上述setUp() 和 tearDown()方法，来实现整个测试类的屏幕录制操作的开启与停止，无须对不同测试场景重复单独构建。

9.5 弹出窗的处理

弹出窗的处理过程包括：通过弹出窗的名称或句柄来定位，切换Driver Context至所需的弹出窗，在弹出窗上执行相关操作步骤，最后跳转回到上级窗口（页面）。

结合我们的测试，创建一个基于浏览器的实例，基于父窗口随后弹出新的窗口，我们统称为子窗口或弹出窗。只要该弹出窗属于当前WebDriver Context，我们都可以对它进行操作。

下图展示一个弹出窗的例子。



创建一个新的测试类**PopupWindowTest**，其中包括**test_popup_window()**方法，代码如下。

```
from selenium import webdriver
import unittest

class PopupWindowTest(unittest.TestCase):

    URL = "https://rawgit.com/upgundecha/learnsewithpython/master/pages/Config.html"

    def setUp(self) :
        self.driver = webdriver.Firefox()
        self.driver.get(self.URL)
        self.driver.maximize_window()
```

```
def test_popup_window(self):
    driver = self.driver

    # save the WindowHandle of Parent Browser Window

    parent_window_id = driver.current_window_handle

    # clicking Help Button will open Help Page in a
new Popup
    # Browser Window
    help_button = driver.find_element_by_id("helpbut
ton")
    help_button.click()
    driver.switch_to.window("HelpWindow")
    driver.close()
    driver.switch_to.window(parent_window_id)

def tearDown(self):
    self.driver.close()

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

在Context调用弹出窗口显示之前，我们先通过current_window_handle属性将父窗口的句柄信息保存下来（稍后我们将使用这个信息从弹出窗返回到父窗口）。接着使用WebDriver下的switch_to.window()方法获取弹出窗的名称或句柄信息，切换到我们要操作的那个弹出窗（子窗口）。

下面我们演示通过名称定位弹出窗。

```
driver.switch_to_window("HelpWindow")
```

我们操作完Help窗口之后，通过close()方法关闭窗口，并且返回至父窗口，代码如下。

```
driver.close()  
  
# switch back to Home page window using the handle  
driver.switch_to_window(default_window)
```

9.6 操作cookies

为了更好的用户体验，cookies作为Web应用一项很重要的手段，将一些诸如用户偏好、登录信息以及各种客户端细节信息，记录并保存在用户计算机本地。WebDriver提供了一组操作cookies的方法，包括读取、添加和删除cookies信息。这些方法可以帮助我们操作cookies，来校验Web应用程序对应的响应。具体方法见下表。

方 法	描 述	参 数	示 例
add_cookie(cookie_dict)	在当前会话中添加cookie信息	cookie_dict:字典对象，包含name与value值	driver.add_cookie({"foo","bar"})
delete_all_cookies()	在当前会话中删除所有cookie信息		driver.delete_all_cookies()
delete_cookie(name)	删除单个名为name的cookie信息	name:要删除的cookie的名称	driver.delete_cookie("foo")
	返回单个名为name的	name:要查找的	

get_cookie(name)	cookie信息。如果没有找到，返回none	cookie的名称	driver.get_cookie("foo")
get_cookies()	返回当前会话所有的cookie信息		driver.get_cookies()

接下的例子，我们来验证用户在首页选择语言后，是否被正确保存至cookie中。

```
import unittest
from selenium import webdriver
from selenium.webdriver.support.ui import Select

class CookiesTest(unittest.TestCase):
    def setUp(self):
        # create a new Firefox session
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://demo.magentocommerce.com/")

    def test_store_cookie(self):
        driver = self.driver
        # get the Your language dropdown as instance of Select class
        select_language = \
            Select(self.driver.find_element_by_id("select-language"))
```

```

        # check default selected option is English
        self.assertEqual("ENGLISH", select_language.first_selected_option.text)
        # store cookies should be none
        store_cookie = driver.get_cookie("store")
        self.assertEqual(None, store_cookie)

        # select an option using select_by_visible_text
        select_language.select_by_visible_text("French")

        # store cookie should be populated with selected country
        store_cookie = driver.get_cookie("store")['value']
        self.assertEqual("french", store_cookie)

    def tearDown(self):
        # close the browser window
        self.driver.quit()

if __name__ == '__main__':
    unittest.main(verbosity=2)

```

上述代码中，我们传递一个cookie的名称，就可以通过get_cookie()方法获取到对应cookie的值。

9.7 章节回顾

在本章，我们学习了一些关于Selenium WebDriver的高级特性，例如键盘和鼠标事件、截屏、（屏幕）录制以及操作cookies。

用ActionChains类模拟各种键盘和鼠标的操作，这在处理大量使用键盘和鼠标操作的应用程序时非常有用。

从测试中，你已经看到了如何运行JavaScript代码。这是一个非常强大的功能特性，让我们能够轻松应对Ajax应用，能够在测试脚本中调用底层的JavaScript API。

当测试过程中产生错误（有可能是测试脚本的问题，也有可能是产品的bug）时，自动截屏或是屏幕录制，都能极大地帮助我们调试测试脚本以及

作为提交bug的重要依据。

最后部分，我们还学习了操作浏览器窗口与cookies的方法。

在下一章节，我们将学习如何将自动化测试与其他持续集成的工具进行联动与整合。

第10章 第三方工具与框架集成

Selenium WebDriver Python API是非常强大和灵活的。到目前为止，我们已经学习了如何通过Selenium WebDriver集成unittest类库来搭建一个简单的自动化测试框架。然而，除了unittest之外，Selenium WebDriver还可以集成很多其他的工具和框架。目前已经有很多基于Selenium WebDriver 实现的框架了。

我们可以通过使Selenium WebDriver与现有的支持BDD（行为驱动开发）的框架结合起来，在自动化测试项目中实现BDD。

还可以将Selenium Python API与持续集成（CI）工具、构建工具相集成，一旦应用程序开发

完成就可以立即执行测试。这可以使开发人员对应用程序的质量和稳定性得到更早的反馈。

本章包含以下一些主要集成的实例：

- 下载和安装Behave；
- 使用Behave编写feature；
- 使用Behave和Selenium WebDriver自动化验证feature；
- 下载和安装Jenkins；
- 搭建Jenkins运行Selenium测试；
- 配置Jenkins捕捉测试结果。

10.1 行为驱动开发（BDD）

BDD是Dan North在他的论文《Introducing BDD》中提出的一种敏捷软件开发方法。

BDD也称为验收测试驱动开发（ATDD）、基于用户故事敏捷测试（story testing）或实例化需求（specification by example）。BDD鼓励软件项目中的开发者、QA 和非技术人员或商业参与者之间的协作，一起定义项目规范，决定验收标准，用自然语言书写出非程序员可读的测试用例。



Python中有很多工具都可以实现BDD，其中两个主要的工具是Behave和Lettuce。

Lettuce是受到了著名的Ruby BDD Cucumber启发。

在接下来的章节中，你将学习到怎样使用Behave来为示例应用程序实现BDD。

10.1.1 Behave安装

安装Behave的过程很简单，通过以下命令可直接下载和安装Behave。

```
pip install behave
```

在命令执行的过程中，会下载并安装Behave和它依赖的第三方包。

10.1.2 第一个feature

在Behave中编写第一个feature的过程从讨论和列举开发中的应用程序的feature和User Story开始。各利益相关者聚在一起，有开发者、测试人员、需求分析师和客户，使用各参与者都能理解的通用语言创建feature、User Story和验收标准的列表。

Behave支持用Given-When-Then（GWT）格式的Gherkin语言创建feature文件。

让我们从示例应用程序中的搜索功能的feature开始。该搜索feature是让用户从主页搜索产品。feature文件以GWT格式对User Story和验收标准进行简单描述，作为一个场景大纲（Scenario Outline），也称为场景步骤（Scenario Steps），解释如下。

- **Given:** 设置一个场景执行的前提条件，本例中——导航到主页。
- **When:** 包含一个场景所要执行的操作，本例中——搜索某件产品。
- **Then:** 包含一个场景执行后的结果，本例中——检查所有匹配的产品列表能否正常显示。

一个场景中可以有多个When和Then。

```
Feature: I want to search for products
```

```
Scenario Outline: Search
```

```
    Given I am on home page  
    when I search for "phone"  
    then I should see list of matching products in search results
```

我们需要将它保存为扩展名为“.feature”的纯文本文件才能在Behave中使用。现在新建一个bdd/feature文件夹，然后将feature文件命名为“search.feature”，保存在该文件夹下。

10.1.2.1 step定义

feature文件完成后，我们需要为feature文件的场景大纲中的step分别进行定义。一个step定义就是一个Python代码块，代码块用简单明了的文字命名，代码调用Python API或者Selenium WebDriver命令来执行该step的内容。step定义文件须保存在feature文件所在路径的子目录“steps”下。下面就创建一个search_steps.py文件来定义上面feature文件中

的step。

```
from behave import *

@given('I am on home page')
def step_i_am_on_home_page(context):
    context.driver.get("http://demo.magentocommerce.com/")

@when('I search for {text}')
def step_i_search_for(context, text):
    search_field = context.driver.find_element_by_name("q")
    search_field.clear()

    # enter search keyword and submit
    search_field.send_keys(text)
    search_field.submit()

@then('I should see list of matching products in search results')
def step_i_should_see_list(context):
    products = context.driver.\
        find_elements_by_xpath("//h2[@class='product-name']/a")
    # check count of products shown in results
    assert len(products) > 0
```

对于每个GWT，我们都需要创建一个匹配的step定义。下面的代码是为Given中的“I am on home page”创建的step定义。使用@修饰符和feature文件

中给出的GWT来标识对应的step，如@given，@when，@then，并且接收一个字符串，该字符串包含对应场景step中剩余部分的步骤描述信息，如本例中的“I am on home page”。

```
@given('I am on home page')
def step_i_am_on_home_page(context):
    context.driver.get("http://demo.magentocommerce.com/")
```

我们也可以将step中内嵌的参数传递给step定义。例如，在@when中我们这么写上面的搜索语句：when I search for“phone”。然后在step定义中通过{text}来读取这个“phone”值，如下代码所示。

```
@when('I search for {text}')
def step_i_search_for(context, text):
    search_field = context.driver.find_element_by_name("q")
    search_field.clear()

    # enter search keyword and submit
    search_field.send_keys(text)
    search_field.submit()
```


从上面代码中就可以看到传递给step定义的上下文变量。Behave通过这个上下文变量来存储要共享的信息。它运行在3个层面，由Behave自动管理。我们还可以使用上下文变量来存储和共享step之间的信息。

10.1.2.2 环境配置

在运行feature之前，需要创建一个环境配置文件，用于配置Behave的常用设置，以及step之间或step定义文件之间的共享代码。如果用Selenium WebDriver和Firefox浏览器执行测试步骤的话，这个文件可以在启动Firefox时初始化WebDriver。下面我们在feature文件目录中新建一个环境配置文件environment.py，添加before_all()和after_all()方法，这两个方法分别在feature执行前和结束后运行。

```
from selenium import webdriver

def before_all(context):
    context.driver = webdriver.Chrome()
```

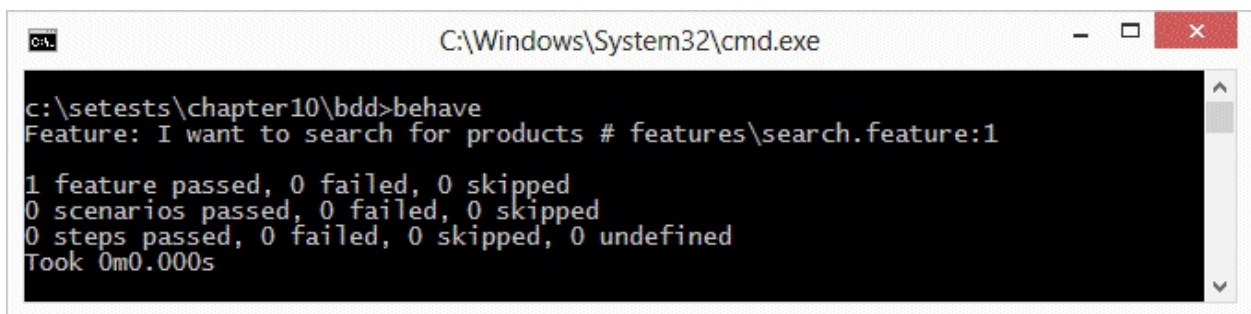
```
def after_all(context):  
    context.driver.quit()
```

10.1.2.3 执行feature

现在可以开始使用Behave执行feature文件了。操作非常简单，从命令行中进入我们之前创建的bdd文件夹目录下，然后执行“behave”命令。

```
behave
```

Behave将执行bdd文件夹下的所有feature文件，通过前面的step定义和环境配置信息来运行相应的scenario。下图是behave命令的执行结果。



```
C:\Windows\System32\cmd.exe  
c:\setests\chapter10\bdd>behave  
Feature: I want to search for products # features\search.feature:1  
  
1 feature passed, 0 failed, 0 skipped  
0 scenarios passed, 0 failed, 0 skipped  
0 steps passed, 0 failed, 0 skipped, 0 undefined  
Took 0m0.000s
```

Behave显示3个级别的结果，即feature、scenario和step各级别通过和失败的数量。

10.1.2.4 使用场景大纲

有时我们可能想用同样的测试步骤、类似数据驱动测试来运行一批已知状态、执行操作和期望结果的测试集合。对于这样的需求，我们可以写出如下的场景大纲。

我们用下面步骤中所给的例子重写 search.feature 文件的场景大纲，这个场景大纲根据“Example”部分中的数据，像模板一样工作。

(1) 本例创建了两个检验搜索功能的例子：根据“类别”搜索和根据“特定产品名”搜索。“Example”部分以表格形式给出搜索项和预期结果。

```
Feature: I want to search for products
```

```
  Scenario Outline: Search
```

```
    Given I am on home page
```

```
      when I search for <term>
```

```
      then I should see results <search_count> in search results
```

Examples: By category

term	search_count
Phones	2
Bags	7

Examples: By product name

term	search_count
Madison earbuds	3

(2) 修改search_steps.py文件以匹配上面的step。

```
from behave import *

@given('I am on home page')
def step_i_am_on_home_page(context):
    context.driver.get("http://demo.magentocommerce.com/")

@when('I search for {text}')
def step_i_search_for(context, text):
    search_field = context.driver.find_element_by_name("q")
    search_field.clear()

    # enter search keyword and submit
    search_field.send_keys(text)
    search_field.submit()

@then('I should see results {text} in search results')
def step_i_should_see_results(context, text):
    products = context.driver.\
```

```
find_elements_by_xpath("//h2[@class='product-name']/a")
# check count of products shown in results
assert len(products) >= int(text)
```

当执行feature时，Behave读取search.feature文件中“Example”部分数据的行数并自动循环执行场景大纲。它将“Example”中的数据传递给scenario中的step，然后执行相应step定义中的命令。下图是Behave执行修改后的feature文件的结果，并打印出了feature上运行的所有组合信息。



Behave也支持用命令行选项 `-junit` 生成junit格式的报吿。

```
C:\Users\amitr\Desktop\setests_final\bdd>behave
Feature: I want to search for products # features\search.feature:1

  Scenario Outline: Search # features\search.feature:3
    Given I am on home page # steps\search_steps.py:4
    When I search for Phones # steps\search_steps.py:9
    Then I should see results 2 in search results # steps\search_steps.py:18

  Scenario Outline: Search # features\search.feature:3
    Given I am on home page # steps\search_steps.py:4
    When I search for Bags # steps\search_steps.py:9
    Then I should see results 7 in search results # steps\search_steps.py:18

  Scenario Outline: Search # features\search.feature:3
    Given I am on home page # steps\search_steps.py:4
    When I search for Madison earbuds # steps\search_steps.py:9
    Then I should see results 3 in search results # steps\search_steps.py:18

1 feature passed, 0 failed, 0 skipped
3 scenarios passed, 0 failed, 0 skipped
9 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m14.539s

C:\Users\amitr\Desktop\setests_final\bdd>
```

10.2 持续集成Jenkins

Jenkins是Java编写的流行的持续集成（CI）服务，起源于Hudson项目。Jenkins和Hudson功能相似。

Jenkins支持各种版本的控制工具，如CVS、SVN、Git、Mercurial、Perforce和ClearCase，而且可以执行用Apache Ant或Java Maven构建的项目。同时，它也可以利用一些插件、shell脚本和Windows批处理命令来构建其他平台的项目。

Jenkins除了构建软件功能外，还可以用于搭建自动化测试环境，实现Selenium WebDriver测试在无人值守的情况下按照预定的时间调度运行，或每次代码变更提交至版本控制系统时实现自动运行的效果。

在接下来的部分，我们将学习如何搭建Jenkins并创建一个自由风格的软件项目来执行测试。

10.2.1 Jenkins环境准备

为了能够成功使用Jenkins执行测试，我们需要做一些修改。我们的目标是在Jenkins上按计划时间执行测试，然后收集测试结果并显示在Jenkins Dashboard页面。为了实现这个目标，我们将重用第2章中创建的冒烟测试。

我们使用了unittest的TestSuite Runner批量执行测试，并以JUnit报告的格式输出测试结果。这就要有xmlrunner的Python库的支持，可以从<https://pypi.python.org/pypi/xmlrunner/1.7.4>获取。

现在执行以下命令下载和安装xmlrunner。

```
pip install xmlrunner
```


冒烟测试脚本smoketests.py是通过TestSuite Runner运行homepagetests.py和searchtest.py两个脚本中的测试的。我们将使用xmlrunner.XML TestRunner来运行冒烟测试并生成JUnit测试报告。此报告以XML格式生成，保存在test-reports子文件夹中。要使用xmlrunner，需要对smoketest.py做些改动，如以下代码的高亮显示部分。

```
import unittest
from xmlrunner import xmlrunner
from searchtest import SearchProductTest
from homepagetests import HomePageTest

# get all tests from SearchProductTest and HomePageTest
class
search_tests = unittest.TestLoader().loadTestsFromTestCase(SearchProductTest)
home_page_tests = unittest.TestLoader().loadTestsFromTestCase(HomePageTest)

# create a test suite combining search_test and home_page_test
smoke_tests = unittest.TestSuite([home_page_tests, search_tests])

# run the suite
xmlrunner.XMLTestRunner(verbosity=2, output='test-reports').run(smoke_tests)
```

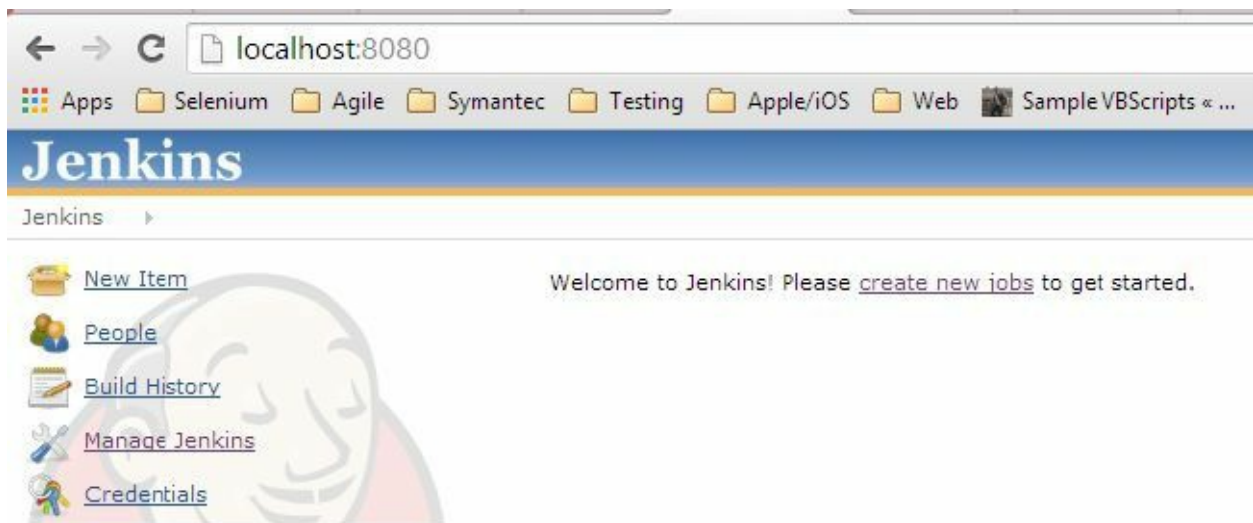
10.2.2 搭建Jenkins

搭建Jenkins相当简单。你可以下载各种平台的Jenkins包并进行安装。在下面的例子中，我们将安装并启动Jenkins，然后创建一个新的构建作业以对示例应用程序进行冒烟测试。

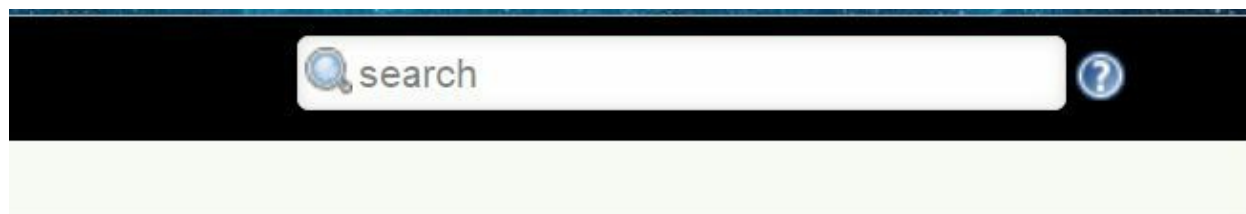
（1）下载并安装Jenkins CI服务器。我们下载的是Jenkins Windows安装包，并在Windows 7上安装Jenkins。

（2）从浏览器中进入**Jenkins Dashboard**页面（默认情况下为[http:// localhost:8080](http://localhost:8080)）。

（3）在**Jenkins Dashboard**页面上，单击新建项目（**New Item**）或创建新作业（**create new jobs**）链接，创建一个新的Jenkins作业，如下图所示。



(4) 在项名称 (**Item name**) 文本框中输入 Demo_App_Smoke_Test，然后选择构建自由风格的软件项目 (**Freestyle project**) 单选按钮，如下图所示。



Item name

☒ **Freestyle project**

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

☐ **Maven project**

Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

☐ **Build multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

☐ **External Job**

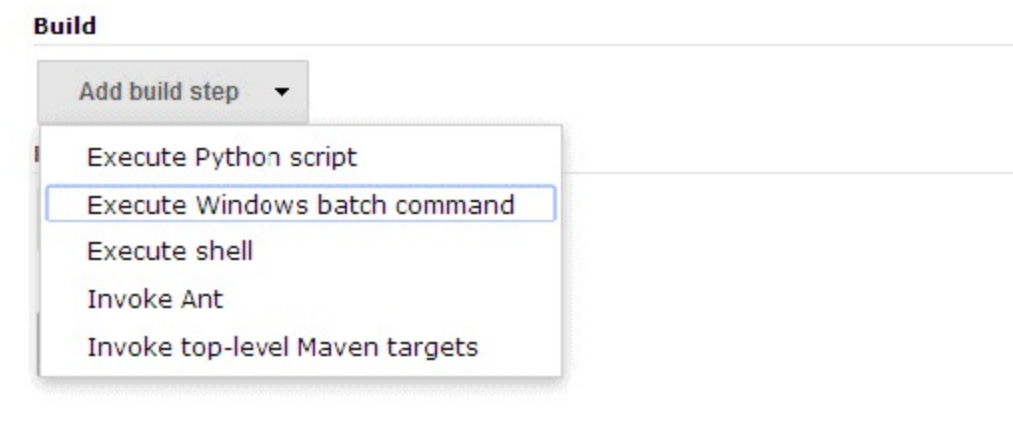
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use

(5) 单击确定（OK）按钮。以上面指定名称命名的新作业就创建成功了。



我们可以连接至各种版本的控制或源代码控制管理（SCM）工具，如SVN、Git、Perforce等，以存储源代码和测试代码。然后作为构建步骤的一部分，获取最新版本的代码，并在Jenkins中构建和测试软件。但是，在本示例中，为了保证过程的简洁性，我们将在执行Windows批处理命令（Execute Windows batch command）构建步骤中将当前文件夹下的测试脚本复制到Jenkins工作空间下，如以下步骤中所述。

（6）在构建（Build）部分中，单击添加构建步骤（Add build step），然后从下拉列表中选择执行Windows批处理命令（Execute Windows batch command）选项，如下图所示。

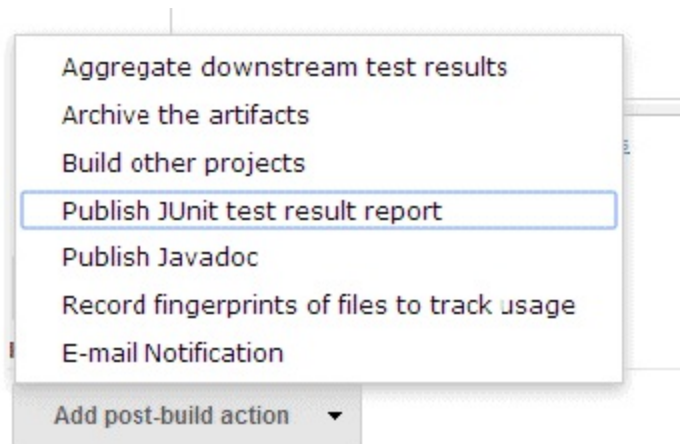


(7) 在命令（Command）文本框中输入以下命令，如下图所示。在不同的电脑上路径可能会有所不同。这个命令将冒烟测试的Python脚本文件复制到Jenkins工作空间下并执行smoketest.py。

```
copy c:\setests\chapter10\smoketests\*.py  
python smoketest.py
```



(8) 我们在前面已经配置了smoketest.py以生成JUnit格式的测试结果，并将测试结果显示在Jenkins Dashboard页面。要在Jenkins中集成这些报告，先单击添加构建后操作（Add post-build action），然后选择发布JUnit测试结果报告（Publish JUnit test result report）选项，如下图所示。



(9) 在构建后操作（Post-build Actions）部分中，在测试报告XML（Test report XMLs）文本框中添加test-reports/ *.xml，如下图所示。Jenkins每次运行测试的时候，它将从test-reports子文件夹中读取测试结果。

Post-build Actions

Publish JUnit test result report

Test report XMLs

[Fileset 'includes'](#) setting that specifies the generated raw XML report files, such as 'myproject/t

☐ Retain long standard output/error

(10) 若想按计划时间自动执行测试，在构建触发器（Build Triggers）部分选择定期构建（Build periodically），并在计划（Schedule）文本框中输入如下图所示数据。那么，每天22点构建过程将自动触发，作为无人值守构建过程的一部分，Jenkins也将自动执行测试，这样第二天早上当你到达办公室的时候就可以看到测试执行结果了。

Build Triggers

☐ Build after other projects are built

☒ Build periodically

Schedule

(11) 单击保存按钮保存作业配置。Jenkins将

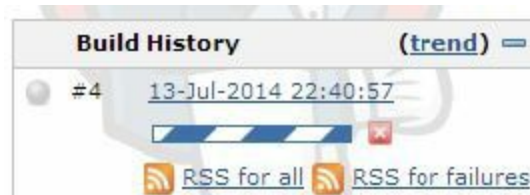
会显示新创建的作业项目页面。

（12）现在可以来检验一下所有的配置项是否设置好，测试是否能成功执行。单击开始构建

（Build Now）链接手动运行该作业，如下图所示。



（13）在构建历史（Build History）部分中可以查看构建的运行状态，如下图所示。



（14）单击构建历史（Build History）部分中正在运行的项目，将打开如下图所示的页面。



（15）除了Jenkins页面上的执行状态和进度条，还可以通过打开控制台输出（Console Output）链接观察后台执行信息。下图是有命令行输出信息的“控制台输出”页面。

Console Output

```
Started by user anonymous
Building in workspace C:\Program Files (x86)\Jenkins\workspace\Demo_App_Smoke_Test_1
[Demo_App_Smoke_Test_1] $ cmd /c call C:\Windows\TEMP\hudson4452624957017176027.bat

C:\Program Files (x86)\Jenkins\workspace\Demo_App_Smoke_Test_1>copy C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final\smoketests\*py
C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final\smoketests\homepagetests.py
C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final\smoketests\searchtests.py
C:\Users\amitr\Desktop\Mrunmayee\Final\setests_final\smoketests\smoketests.py
3 file(s) copied.

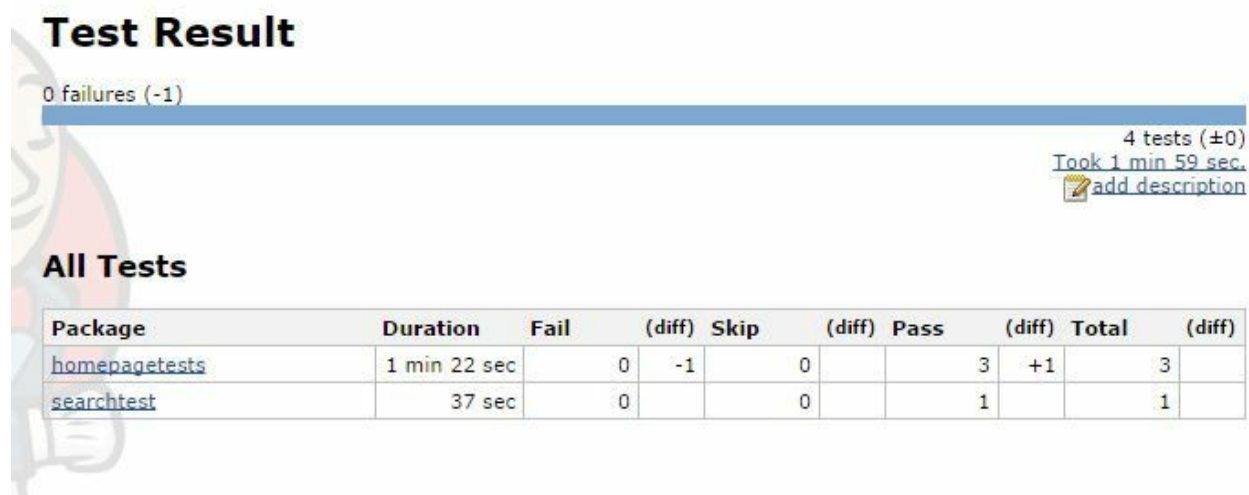
C:\Program Files (x86)\Jenkins\workspace\Demo_App_Smoke_Test_1>python smoketests.py
Build was aborted
Aborted by anonymous
Recording test results
ERROR: Publisher hudson.tasks.junit.JUnitResultArchiver aborted due to exception
hudson.AbortException: No test report files were found. Configuration error?
    at hudson.tasks.junit.JUnitParser$ParseResultCallable.invoke(JUnitParser.java:116)
    at hudson.tasks.junit.JUnitParser$ParseResultCallable.invoke(JUnitParser.java:92)
    at hudson.FilePath.act(FilePath.java:981)
    at hudson.FilePath.act(FilePath.java:959)
    at hudson.tasks.junit.JUnitParser.parseResult(JUnitParser.java:89)
    at hudson.tasks.junit.JUnitResultArchiver.parse(JUnitResultArchiver.java:121)
    at hudson.tasks.junit.JUnitResultArchiver.perform(JUnitResultArchiver.java:138)
    at hudson.tasks.BuildStepCompatibilityLayer.perform(BuildStepCompatibilityLayer.java:74)
    at hudson.tasks.BuildStepMonitor$1.perform(BuildStepMonitor.java:20)
    at hudson.model.AbstractBuild$AbstractBuildExecution.perform(AbstractBuild.java:770)
    at hudson.model.AbstractBuild$AbstractBuildExecution.performAllBuildSteps(AbstractBuild.java:734)
    at hudson.model.Build$BuildExecution.post2(Build.java:183)
    at hudson.model.AbstractBuild$AbstractBuildExecution.post(AbstractBuild.java:683)
    at hudson.model.Run.execute(Run.java:1784)
    at hudson.model.FreeStyleBuild.run(FreeStyleBuild.java:43)
    at hudson.model.ResourceController.execute(ResourceController.java:89)
    at hudson.model.Executor.run(Executor.java:240)
Finished: ABORTED
```

(16) 一旦Jenkins完成构建过程，就可以看到
一个类似于下一个截图所示的构建页面。

(17) Jenkins通过读取unittest框架生成的测试结果文件，在页面上显示测试结果和其他各项指标。单击构建页面上的测试结果（Test Results）链接可以查看Jenkins保存的测试结果。

(18) 我们之前配置的测试结果以JUnit格式生

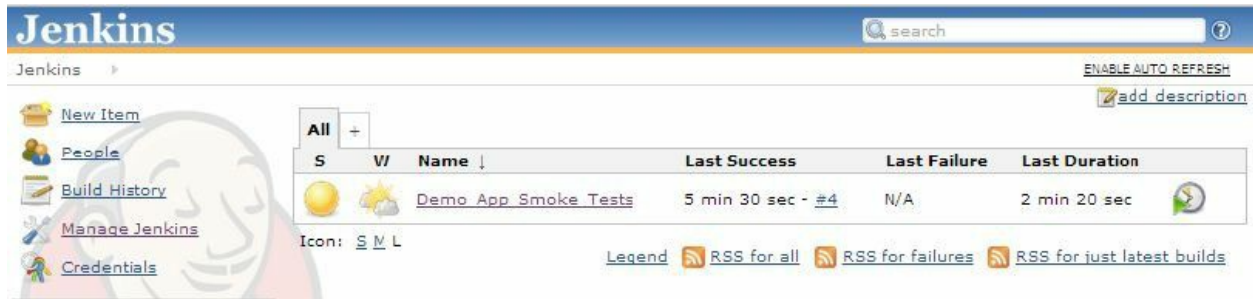
成。当单击测试结果（Test Results）时，Jenkins将会显示JUnit测试结果，如下图所示，显示测试结果摘要，其中失败的测试会高亮显示。






（19）我们也可以单击Package名字的连接来查看各个测试的详细结果信息，如下图所示。



Jenkins还会以下图所示的格式在Dashboard主页上显示所有构建作业的最终状态信息。



The screenshot shows the Jenkins Dashboard interface. At the top, there's a blue header with the 'Jenkins' logo and a search bar. Below the header, a navigation sidebar on the left contains links for 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. The main content area displays a table of build jobs. The table has columns for 'S' (Status), 'W' (Warnings), 'Name', 'Last Success', 'Last Failure', and 'Last Duration'. A single row is visible for a job named 'Demo App Smoke Tests', which is in a successful state (yellow ball icon) and has a duration of '2 min 20 sec'. Above the table, there are controls for 'All' builds and a '+'. Below the table, there are links for 'Icon: S W L' and 'Legend', along with RSS feeds for 'all', 'failures', and 'latest builds'. The top right of the dashboard includes a 'search' bar, a 'Jenkins' dropdown, and links for 'ENABLE AUTO REFRESH' and 'add description'.

S	W	Name ↓	Last Success	Last Failure	Last Duration
		Demo App Smoke Tests	5 min 30 sec - #4	N/A	2 min 20 sec 

10.3 章节回顾

在本章中，我们学习了如何将Selenium与BDD（Behave）和CI（Jenkins）集成。实现了将Selenium WebDriver API与Behave集成，并通过编写feature和step定义文件来执行自动化验收测试。

通过搭建Jenkins运行Selenium WebDriver测试，从而实现每晚在无人值守的情况下自动构建程序和执行测试。Jenkins提供了一个易于搭建的平台，来对接各种程序开发平台的构建和运行测试作业。

到目前为止，我们已经完成了Python Selenium WebDriver的学习之旅。我们主要学习了使用Selenium WebDriver进行Web应用程序在浏览器中的交互测试和自动化测试的基本知识点，运用这些知识点，我们就可以构建自己的自动化测试框架

了。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

新年新气象

社区UI全新改版，崭新面貌迎接2017！为答谢社区用户，

即日起
1月26号

全场电子书8折优惠！



前端开发



数据科学



编程语言



移动开发



游戏开发

机器学习&深度学习

更多>>



Python机器学习——预测分析核心算法



贝叶斯方法：概率编程与贝叶斯推断



机器学习项目开发实战



贝叶斯思维：统计建模的Python学习法

免费电子书
Free eBook

立即领取

我要写书
Write for Us

立即查看

近期活动

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。


与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“**57AWG**”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



Wireshark网络分析的艺术

作者：林沛满
责编：傅道坤
分类：计算机科学 > 安全与加密 > 网络安全

Wireshark是当前最流行的网络包分析工具。它上手简单，无需培训就可入门。很多棘手的网络问题遇到Wireshark都能迎刃而解。本书挑选的网络包来自真实场景，经典且接地气。讲解时采用了生活化的

[下载PDF样章](#) [配套文件下载](#)

分享： 

浏览 5.6K 想读 57 推荐 7

纸质 ¥45.00-¥31.50 (7折) 电子 ¥25.00 电子+纸质 ¥45.00

[购买](#)



(纸质)



(纸质)

总价: 75.60

[一起购买](#)

[目录](#) [评论 9](#) [勘误 1](#) [出版信息](#)

[作者简介](#) [专业书评](#) [内容提要](#)

本书作者



LinPeiman
上海
1.0K经验值

[发私信](#) [送积分](#) [关注](#)

《Wireshark网络分析就这么简单》即《Wireshark网络分析的艺术》作者

兑换样书

[立即兑换](#)

[如何赚取积分](#)

电子书版本

[PDF](#) [Epub](#) [Mobi](#)

精彩推荐



Nmap渗透测试指南
作者：商广明

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在这里一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：436746675

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询：contact@epubit.com.cn